MICROSOFT®

The High Performance Software™

CodeView

Microsoft® CodeView™

Window-Oriented Debugger

for the MS-DOS® Operating System

Microsoft Corporation

Information in this document is subject to change without notice and does not represent a commitment on the part of Microsoft Corporation. The software described in this document is furnished under a license agreement or nondisclosure agreement. The software may be used or copied only in accordance with the terms of the agreement. It is against the law to copy this software on magnetic tape, disk, or any other medium for any purpose other than the purchaser's personal use.

© Copyright Microsoft Corporation, 1986

If you have comments about the software, complete the Software Problem Report at the back of this manual and return it to Microsoft Corporation.

If you have comments about the software documentation, complete the Documentation Feedback reply card at the back of this manual and return it to Microsoft Corporation.

Microsoft, the Microsoft logo, MS, MS-DOS, and XENIX are registered trademarks of Microsoft Corporation. CodeView and The High Performance Software are trademarks of Microsoft Corporation.

IBM is a registered trademark of International Business Machines Corporation.

Intel is a registered trademark of Intel Corporation.

UNIX is a trademark of AT&T Bell Laboratories.

Document Number 410840010-400-R00-0486
Part Number 048-014-036

Contents

1 Introduction to the Microsoft® CodeView™ Debugger 3

1.1 Introduction 5
1.2 Overview 5
1.3 About This Manual 6
1.4 Notational Conventions 8

2 Getting Started 13

2.1 Introduction 15
2.2 Starting the Sample Session 15
2.3 Preparing C Programs for the CodeView Debugger 16
2.4 Starting the CodeView Debugger 19
2.5 Using CodeView Options 23
2.6 Using the CodeView Debugger with the Macro Assembler 32

3 The CodeView Display 33

3.1 Introduction 35
3.2 Using Window Mode 36
3.3 Using Sequential Mode 63

4 Using Dialog Commands 65

4.1 Introduction 67
4.2 Entering Commands and Arguments 67
4.3 Format for CodeView Commands and Arguments 69
4.4 C Expressions 70

5 Executing Code 79

5.1 Introduction 81
5.2 Trace Command 82
5.3 Program Step Command 84
5.4 Go Command 87
5.5 Execute Command 90
5.6 Restart Command 91

6 Examining Data and Expressions 93

6.1 Introduction 95
6.2 Display Expression Command 95
6.3 Examine Symbols Command 100
6.4 Dump Commands 103
6.5 Register Command 113
6.6 8087 Command 115

7 Managing Breakpoints 119

7.1 Introduction 121
7.2 Breakpoint Set Command 121
7.3 Breakpoint Clear Command 124
7.4 Breakpoint Disable Command 125
7.5 Breakpoint Enable Command 127
7.6 Breakpoint List Command 128

8 Managing Watch Statements 131

8.1 Introduction 133
8.2 Setting Watch-Expression and Watch-Memory Statements 134
8.3 Setting Watchpoints 138
8.4 Setting Tracepoints 141
8.5 Deleting Watch Statements 146
8.6 Listing Watchpoints and Tracepoints 148

9 Examining Code 151

9.1 Introduction 153
9.2 Set Mode Command 153
9.3 Unassemble Command 155
9.4 View Command 158
9.5 Current Location Command 161
9.6 Stack Trace Command 162

10 Modifying Code or Data 165

10.1 Introduction 167
10.2 Assemble Command 167
10.3 Enter Commands 170
10.4 Register Command 181

11 Using System-Control Commands 185

11.1 Introduction 187
11.2 Help Command 187
11.3 Quit Command 188
11.4 Radix Command 189
11.5 Redraw Command 191
11.6 Screen Exchange Command 191
11.7 Search Command 192
11.8 Shell Escape Command 195
11.9 Tab Set Command 198
11.10 Redirection Commands 199

CodeView Appendixes 207

A Command and Mode Summary 209

A.1 Introduction 211
A.2 Modes 211
A.3 Options 212
A.4 Window Commands 213
A.5 Dialog Commands 215
A.6 Type Specifiers 218

B Regular Expressions 221

B.1 Introduction 223
B.2 Special Characters in Regular Expressions 223
B.3 Searching for Special Characters 224
B.4 Using the Period 224
B.5 Using Brackets 225
B.6 Using the Asterisk 226
B.7 Matching the Start or End of a Line 227

C Error Messages 229

Glossary 237

Index 247

Figures

Figure 1.1 CodeView Screen in Window Mode 11

Figure 2.1 CodeView Start-Up Screen in Window Mode 22

Figure 3.1 Elements of the CodeView Debugging Screen 36

Figure 3.2 The File Menu 49

Figure 3.3 The Search Menu 51

Figure 3.4 The View Menu 53

Figure 3.5 The Run Menu 54

Figure 3.6 The Watch Menu 55

Figure 3.7 The Options Menu 57

Figure 3.8 The Calls Menu 60

Figure 8.1 Watch-Command Statements in the Watch Window 137

Figure 8.2 Watchpoint-Command Statements in the Watch Window 140

Figure 8.3 Tracepoints in the Watch Window 145

Tables

Table 2.1 Default Exchange and Display Modes 28

Table 4.1 CodeView Operators 70

Table 4.2 CodeView Radix Examples 74

Table 4.3 Registers 74

Table 6.1 Types for printf 96

Table 10.1 Flag-Value Mnemonics 182

Table A.1 CodeView Modes 211

Table A.2 Window Commands 213

Table A.3 Menu Selections 214

Table A.4 Dialog Commands 216

Table A.5 Type Specifiers 219

Chapter 1

Introduction to the Microsoft® CodeView™ Debugger

1.1 Introduction 5

1.2 Overview 5

1.3 About This Manual 6

1.4 Notational Conventions 8

1.1 Introduction

The Microsoft® CodeView™ debugger is a debugging program that helps you test executable files developed with the Microsoft C Compiler. This chapter introduces you to CodeView debugger, and summarizes the manual and the conventions used in it.

1.2 Overview

The CodeView debugger can display and execute program code, control program flow, and examine or change values in memory. Its window interface makes debugging easy. You can view your source code in one window, commands and responses in another, registers and flags in a third, and the values of variables or expressions in a fourth. You can examine the values of global or local variables, either by themselves or combined with other variables in expressions.

The window interface is designed for IBM® Personal Computers and IBM-compatible computers. However, you can also use the CodeView debugger with non-IBM-compatible computers using a sequential interface. Any debugging operation that can be performed with the window interface can also be performed with the sequential interface.

The CodeView debugger can access program locations through addresses, symbols, or line-number references. This makes it easy to locate and debug specific sections of code. You can debug programs at the source level, or you can examine code at the machine level in the debugger's assembly-language mode.

CodeView commands can be entered either from the keyboard or, in many cases, with the Microsoft Mouse (with the window interface only). Once you learn the commands, you can work most efficiently using both the mouse and the keyboard. The mouse is not required; all commands can be entered from the keyboard.

Note

The CodeView debugger is designed specifically for the Microsoft Mouse. Many manufacturers advertise their pointing devices as being compatible with the Microsoft Mouse. The CodeView debugger may work with some of these devices if they are closely compatible.

The CodeView debugger is simple to learn and use. Its commands are logical and easy to understand, especially for programmers who are familiar with the Microsoft Symbolic Debugging Utility (**SYMDEB**) or the **DEBUG** utility provided with MS-DOS®. The CodeView user interface shares some features with its predecessors, but also incorporates powerful new features such as popup menus, multiple windows, mouse support, and single-keystroke commands.

1.3 About This Manual

This manual explains how to use the CodeView debugger to examine programs and locate program errors. It is a companion manual to the *Microsoft C Compiler User's Guide*, the *Microsoft C Compiler Language Reference*, and the *Microsoft C Compiler Run-Time Library Reference*. See those manuals for information about developing C programs.

Although the manual focuses on debugging C programs using source code, the CodeView debugger can also debug assembly-language programs, or it can debug C programs at the assembly-language level. If you are not familiar with assembly-language programming, some aspects of the debugger may be unfamiliar (assembly mode and the register window, in particular).

However, all CodeView features relating to assembly language are optional. You can simply ignore the assembly-mode features and debug your programs in the C source mode. If you wish to learn more about assembly-language programming, you should consider purchasing the Microsoft Macro Assembler and reading its manual along with one of the many tutorial books on assembly-language programming.

The following list tells where to find information on various aspects of CodeView:

For This Information:	See:
Compiling and linking C programs in the special format required by the CodeView debugger, and invoking the debugger with various command-line options	Chapter 2, "Getting Started"
Using elements of the CodeView display, including windows, popup menus, and the mouse	Chapter 3, "The CodeView Display"
Specifying arguments for dialog commands and using the CodeView operators to create expressions	Chapter 4, "Using Dialog Commands"
Executing all or part of your program	Chapter 5, "Executing Code"
Testing the value of expressions, or examining data of different sizes	Chapter 6, "Examining Data and Expressions"
Setting, enabling, disabling, clearing, and listing breakpoints	Chapter 7, "Managing Breakpoints"
Creating watch statements and managing the watch window	Chapter 8, "Managing Watch Statements"
Examining code and tracing function or procedure calls	Chapter 9, "Examining Code"
Modifying data or code in memory	Chapter 10, "Modifying Code or Data"
Controlling the operation of the CodeView debugger	Chapter 11, "Using System-Control Commands"

In addition to the information above, the following information is included in the appendixes:

For This Information:	See:
A summary of CodeView modes, commands, and menus	Appendix A, "Command and Mode Summary"

How to use regular expressions to find variable text strings in a source file	Appendix B, "Regular Expressions"
A list of CodeView error messages	Appendix C, "Error Messages"
Definitions of terms used in the manual	Glossary

Important

There may be additional information about the CodeView debugger in the **README.DOC** file provided on your Microsoft C Compiler distribution disk. This file will describe any additions to the documentation or changes made to the program after the manual was printed.

1.4 Notational Conventions

The following notational conventions are used throughout this manual:

Convention	Meaning
Bold	Bold type indicates text that must be typed as shown. Text that is shown in bold type includes the following: operators, keywords, and standard functions. Examples are shown below: **+=** **_setargv** **if** **sizeof** **main** **int**
BOLD CAPITALS	Bold capital letters are used for registers, environment variables, and the names of executable files and files provided with the product. Commands typed at the MS-DOS level are also capitalized. These commands include built-in MS-DOS commands such as **SET**, as well as program names such as **CV**. You are not required to use capital letters when you actually enter these commands.

Italics

Italics mark placeholders in command lines and option specifications. A placeholder represents a variable item that must appear at a specific point. Consider the command syntax for the Radix command:

N*number*

Note that *number* is italicized to indicate that it represents a general form for the Radix (N) command. In an actual command, the user supplies a particular number for the placeholder *number*.

Occasionally, italics may be used to emphasize particular words in the text.

Examples

Examples are displayed in a special typeface so that they will look more like the programs you create with a text editor or the output of commonly used computer printers.

User Input

If a command produces output, the input that you type is shown in boldface, while the output displayed by the CodeView debugger is shown in regular, nonboldface type, as in the following example:

```
>RAX
AX 0041
:43
>
```

Ellipsis dots

Vertical ellipsis dots are used in program examples to indicate that a portion of the program has been omitted. For instance, in the following excerpt, three statements are shown. The ellipsis dots between the statements indicate that intervening program lines occur, but are not shown.

```
count = 0;
.
.
.
*pc++;
.
.
.
count = 0;
```

[Double brackets]

Double brackets enclose optional fields in command-line and option syntax. Consider the following command-line syntax:

R [[*register*]] [[[=]]*value*]]

The double brackets around the placeholders indicate that you may enter a *register* and you may enter a *value*. The equal sign (=) in the second set of brackets indicates that you may place an equal sign before the *value*, but only if you specify a *value*.

Single brackets are used to indicate brackets used by C array declarations and subscript expressions. For instance, `a[10]` is an example of brackets in a C subscript expression.

Vertical bar

The vertical bar indicates that you may enter one of the entries shown on either side of the bar. The following syntax block illustrates a vertical bar:

DB [[*address* | *range*]]

The bar indicates that following the Dump Bytes command (**DB**), you can specify either an *address* or a *range*. Since both are in double brackets, you can also give the command with no argument.

"Quotation marks"

Quotation marks set off terms defined in the text. For example, the term "highlight" appears in quotation marks the first time it is defined.

Some C constructs require quotation marks. Quotation marks required by the language have the form "" rather than " ". For example, a C string used in an example would be shown in the following form:

```
"abc"
```

SMALL CAPITALS

Small capital letters are used for the names of keys and key sequences, such as ENTER, CONTROL-C, and ALT-F.

Sample screens

Sample screens are shown in black and white. Your screens will look like this if you have a monochrome monitor, or if you use the **/B** option in the CodeView command line (see Section 2.5.1, "Starting with a Black-and-White Display," for more information). Figure 1.1 shows an example. Screens will be slightly different if you use a color monitor in color mode.

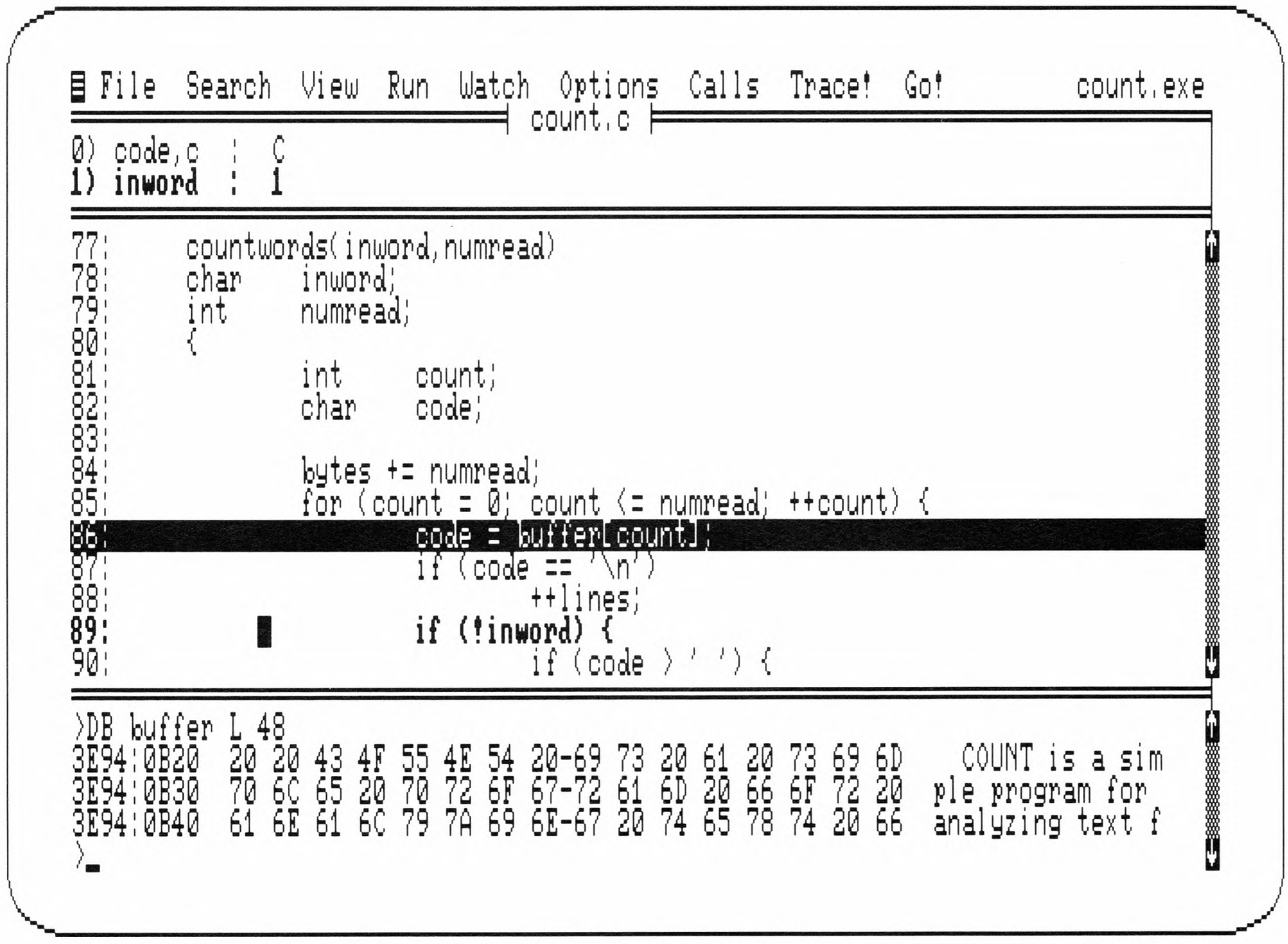


Figure 1.1 CodeView Screen in Window Mode

Chapter 2
Getting Started

2.1 Introduction 15

2.2 Starting the Sample Session 15

2.3 Preparing C Programs for the CodeView Debugger 16

2.3.1 Writing C Source Code 16

2.3.2 Compiling Source Files for the CodeView Debugger 17

2.3.3 Linking Object Files for the CodeView Debugger 18

2.4 Starting the CodeView Debugger 19

2.5 Using CodeView Options 23

2.5.1 Starting with a Black-and-White Display 25

2.5.2 Specifying Start-Up Commands 26

2.5.3 Setting the Screen-Exchange Mode 27

2.5.4 Enabling Window or Sequential Mode 29

2.5.5 Turning Off the Mouse 30

2.5.6 Using the Enhanced Graphics Adapter's 43-Line Mode 31

2.6 Using the CodeView Debugger with the Macro Assembler 32

2.1 Introduction

Getting started with the CodeView debugger requires several simple steps. You must prepare a special-format executable file for the program you wish to debug; then you invoke the debugger. You may also wish to specify options that will affect the operation of the debugger.

This chapter describes a disk-based tutorial session that introduces you to the CodeView debugger. The chapter also describes how to compile and link a C program to produce an executable file in the CodeView format, and how to load a program into the CodeView debugger. Using the debugger with assembly-language programs is also discussed briefly.

2.2 Starting the Sample Session

The sample session provided on the demonstration disk illustrates many reatures of the CodeView debugger. You may wish to try this session before reading further in the manual.

To start the session, log on to the drive containing the sample disk and type `SAMPLE`. This starts the batch file **SAMPLE.BAT**, which displays explanatory text describing the purpose of the session.

During the session, the batch file automatically loads a program into the debugger. Debugging commands are then redirected from a text file into the debugger. Throughout the session, comments will tell you what is happening and why. Periodically, you will be asked to press a key to continue; this allows you to control the pace of the session.

Note

If you wish to quit before the sample session is finished, press CONTROL-C or CONTROL-BREAK. If you are still in the batch file, a prompt appears asking if you want to terminate. Enter `Y` for yes. If the CodeView debugger has been started, the work `break` appears, followed by the CodeView prompt (>). Enter `Q` (the Quit command.) This returns you to the MS-DOS operating system.

2.3 Preparing C Programs for the CodeView Debugger

Executable files must be in a special format in order to be used with the CodeView debugger. The special-format files contain line-number information and a symbol table in addition to executable code. You must use the correct options to put this additional information into the object files during compilation, and then into the executable file during linking.

If you try to debug an executable file that does not contain this additional information, the debugger will not be able to interpret symbols or to correlate code addresses and source line numbers. You can still debug your program in assembly mode, but many of the most powerful features of the CodeView debugger will be disabled.

You must compile your C programs with one of the two compilation-control programs provided with the Microsoft C Compiler: **MSC** or **CL**. You must link your program with the Microsoft Overlay Linker, **LINK**. See the *Microsoft C Compiler User's Guide* for complete instructions on compiling and linking. The special steps required for compiling and linking programs for the CodeView debugger are explained in the sections 2.3.1–2.3.3.

2.3.1 Writing C Source Code

Any source code that is legal in C can be compiled to an executable file and debugged with the CodeView debugger. However, some programming practices make debugging more difficult. You should be aware of how these practices affect debugging.

The CodeView debugger will be easier to use if you put each source statement on a separate source line. For example, the following code is legal in C:

```
code = buffer[count]; if (code == '\n') ++lines;
```

This code is actually made up of three separate C statements. When they are placed together on the same line, the separate statements cannot be accessed individually during source debugging. You would not be able to put a breakpoint on the statement `++lines;` or execute to the statement `if (code == '\n')`. (A breakpoint is an address that stops program execution each time the address is encountered.) The same code would be easier to debug if it were written in the following form:

```
code = buffer[count];
if (code == '\n')
        ++lines;
```

This also makes code easier to read and corresponds with what is generally considered good programming practice.

Macros cannot be easily debugged with the CodeView debugger. This is not a problem with most simple macros, but if you have complex macros with potential side effects, you may need to write them first as regular source statements. After they are debugged, you can put them in macros.

The C language also permits you to put code in separate include files and then read the files into your source file using the #**include** directive. However, you will not be able to use the CodeView debugger to debug source code in include files. Therefore, it is better to use include files for macros only. The preferred method of using standard code in C is to write and compile separate library modules, then link the resulting object files with your programs. The CodeView debugger does support this technique.

2.3.2 Compiling Source Files for the CodeView Debugger

When you compile a source file for a program you want to debug, you must specify the /**Zi** option in response to the **MSC** prompts, or in an **MSC** or **CL** command line. The /**Zi** option instructs the compiler to include line-number and symbol information in the object file.

If you do not need complete symbolic information in some modules, you can compile those modules with the /**Zd** option instead of /**Zi**. The /**Zd** option writes less symbolic information to the object file, so using this option will save disk space and memory. For example, if you are working on a program made up of five modules, but you only need to debug one module, you can compile that module with the /**Zi** option and the other modules with the /**Zd** option. The completed executable file will be significantly smaller than if you had compiled all options with /**Zi**. You will be able to examine global variables and see source lines in modules compiled with the /**Zd** option, but local variables will be unavailable.

In addition, you will probably want to specify the /**Od** option. This option turns off optimization. Optimized code may be rearranged for greater efficiency and, as a result, the instructions in your program may not correspond closely to the source lines. After debugging, you can compile a final version of the program with the optimization level you prefer.

Note that you cannot debug a program until you compile it successfully. The CodeView debugger will not help you correct syntax or compiler errors. You should refer to a source listing and the appropriate reference books to correct your code until you can compile it successfully; then use the debugger to locate logical errors in the program. Finally, you can correct these errors in the source code and recompile.

■ Example

```
MSC COUNT /Zi /Od;
```

This example compiles the source file **COUNT.C** to produce an object file called **COUNT.OBJ**. The object file contains line-number information, a symbol table, and unoptimized object code.

2.3.3 Linking Object Files for the CodeView Debugger

When you link an object file or files for debugging, you should specify the **/CODEVIEW** option (it can be abbreviated to **/CO**). This instructs the linker to incorporate addresses for symbols and source lines into the executable file.

No other option is necessary for the debugger, but you may use other options if your program requires them (see the *Microsoft C Compiler User's Guide*). For example, you could use the **/MAP** or **/PAUSE** option.

Warning

You should not use the **/EXEPACK** option with the CodeView debugger, since this option strips all symbolic information from the executable file. The **EXEPACK** utility also strips symbolic information. If the debugger detects a packed file on start-up, you will see a warning message. You can still attempt to debug the file in assembly mode, but there will be no symbolic information.

Although executable files prepared with the **/CODEVIEW** option can be executed from the MS-DOS command line like any other executable files, they are larger because of the extra symbolic information in them. When you finish debugging a program, you will probably want to link your final version without the **/CODEVIEW** option to minimize program size.

■ Examples

```
LINK /CO COUNT;

CL /Zi /Od COUNT.C
```

The first example links the object file **COUNT.OBJ** to form an executable file, **COUNT.EXE**, containing the symbolic and line-number information required by the CodeView debugger.

The second example uses the **CL** command to compile and link an executable file in one operation. When you use the **CL** command, you do not need to specify the **/CODEVIEW** link option, since the program automatically supplies this option when you specify the **/Zi** compile option.

2.4 Starting the CodeView Debugger

Before starting the debugger, make sure all the files it requires are available in the proper places. The following files are recommended for C source debugging:

File	Location
CV.EXE	The CodeView program file can be in the current directory, or in any directory accessible with the **PATH** command. For example, if you set up your C compiler files according to the suggestions in the *Microsoft C Compiler User's Guide*, you could put **CV.EXE** in the **BIN** directory.
CV.HLP	If you want to have the on-line help available during your session, you should have this file either in the current directory, or in any directory accessible with the **PATH** command. For example, if you set up your C compiler files according to the suggestions in the *Microsoft C Compiler User's Guide*, you could put **CV.HLP** in the **BIN** directory. If the CodeView debugger cannot find the help file, you can still use the debugger, but you will see an error message if you try to use one of the help commands.

program.**EXE**	The executable file for the program you wish to debug must be in the current directory, or in a drive and directory you specify as part of the start-up file specification. The CodeView debugger will display an error message and refuse to start if the executable file is not found.
program.**C**	Normally source files should be in the current directory. However, if you give a file specification for the source file during compilation, that specification will become part of the symbolic information stored in the executable file. For example, if you compiled with the command line `MSC DEMO`, the CodeView debugger will expect the source file to be in the current directory. However, if you compiled with the command line `MSC \C\DEMO`, then the debugger will expect the source file to be in directory `\C`. If the debugger cannot find the source file in the directory specified by the executable file (usually the current directory), the program will prompt you for a new directory. You can either enter a new directory, or you can press the ENTER key to indicate that you do not want a source file to be used for this module. If no source file is specified, you must debug in assembly mode.

If the appropriate files are in the correct directories, you can enter the CodeView command line at the MS-DOS command prompt. The command line has the following form:

CV [[*options*]] *executablefile* [[*arguments*]]

The *options* are one or more of the options described in Section 2.5. The *executablefile* is the name of an executable file to be loaded by the debugger. It must have the extension **.EXE** or **.COM**. If you try to load a nonexecutable file, the following message appears:

```
Not an executable file
```

C programs and assembly-language programs containing CodeView symbolic information will always have the extension **.EXE**. Files with the extension **.COM** can be debugged in assembly mode, but they can never contain symbolic information. Programs that use overlays cannot be debugged with the CodeView debugger.

The optional *arguments* are parameters passed to the *executablefile*. If the program you are debugging does not accept command-line arguments, you do not need to pass any arguments.

If you specify the *executablefile* as a file name with no extension, the CodeView debugger searches for a file with the given base name and the extension **.EXE**. If the file you specify is not in the CodeView format, the debugger starts in assembly mode and displays the following message:

```
No symbolic information
```

You must specify an executable file when you start the CodeView debugger. If you omit the executable file, the debugger displays a message showing the correct command-line format.

When you give the debugger a valid command line, the executable program and the source file are loaded, the address data is processed, and the CodeView display appears. The initial display will be in window mode or sequential mode, depending on the options you specify and the type of computer you have.

For example, if you wanted to debug the program **SIEVE.EXE**, you could start the debugger with the following command line:

```
CV SIEVE
```

If you give this command line on an IBM Personal Computer, window mode will be selected automatically. The display will look like Figure 2.1, shown on the following page.

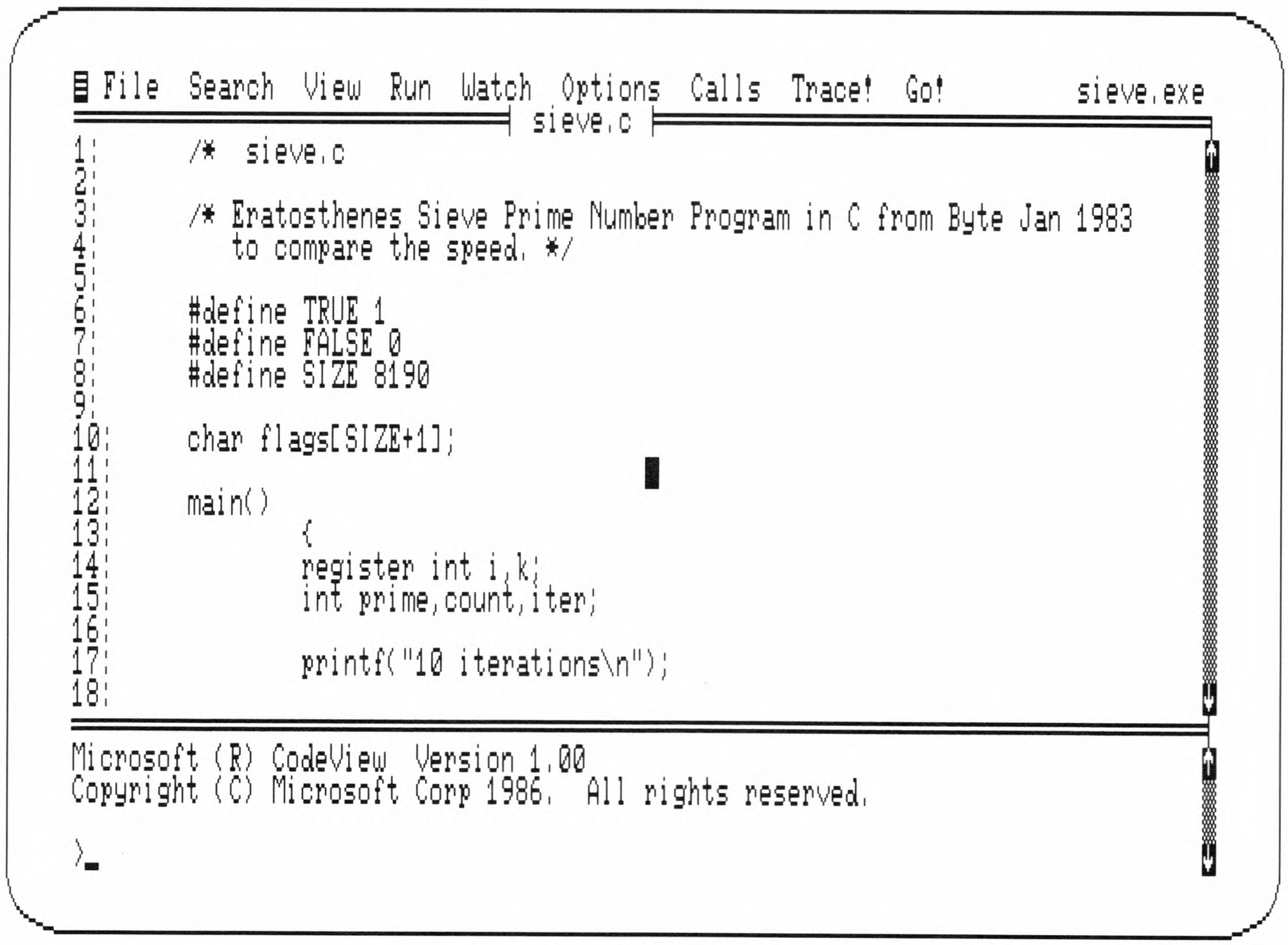


Figure 2.1 CodeView Start-Up Screen in Window Mode

If you give the same command line on most non-IBM computers, sequential mode will be selected. The following lines appear:

```
Microsoft (R) CodeView  Version 1.00
Copyright (C) Microsoft Corp 1986. All rights reserved.

>
```

You can use CodeView options, as described in Section 2.5, to override the default start-up mode.

If your program is written in C, the CodeView debugger is now at the beginning of the C start-up code that precedes your program. In source mode, you can enter an execution command (such as Trace or Program Step) to automatically execute through the start-up code to the beginning of your program. At this point, you are ready to start debugging your program as described in chapters 3 through 11.

2.5 Using CodeView Options

You can change the start-up behavior of the debugger by specifying options in the command line.

An option is a sequence of characters preceded by either a forward slash (/) or a dash (–). A file containing a dash in its file name must be renamed before use with CodeView so that the debugger will not interpret the dash as an option designator. You can use more than one option in a command line, but each option must have its own option designator and spaces must separate each option from other elements of the command line.

Note

The CodeView debugger's defaults are different for IBM Personal Computers than for other computers. However, the debugger may not always recognize the difference between computers. The debugger determines if you have an IBM computer by looking at certain locations in memory, and by checking for PC-DOS, the IBM version of MS-DOS. If you use PC-DOS on certain IBM-compatible computers, the CodeView debugger may think you have an IBM computer and act accordingly.

The list on the following page suggests some situations in which you might want to use an option. If more than one condition applies, you can use more than one option (in any order). If none of the conditions applies, do not use any options.

If:	Type:
You have an IBM-compatible computer and you want to use window mode	**/W**
You have a two-color monitor, a color graphics adapter (CGA), and an IBM or IBM-compatible computer	**/B**
You are debugging a graphics program and you want to be able to see the output screen	**/S**
You are debugging a program that uses multiple video-display pages and you want to be able to see the output screen	**/S**
You are using a non-IBM-compatible computer and you want to be able to see the output screen	**/S**
You are using an IBM-compatible computer to debug a program that does not use graphics or multiple video-display pages and you want to be able to see the output screen	**/F**
You have an IBM computer, but you wish to debug in sequential mode (for example, with redirection)	**/T**
You have a mouse installed in your system, but you do not want to use it during the debugging session	**/M**
You want a 43-line display and you have an IBM or IBM-compatible computer with an enhanced graphics adapter (EGA)	**/43**
You want the CodeView debugger to automatically execute a series of commands when it starts up	**/C***commands*

For example, assume you are using an IBM-compatible computer with a CGA and a two-color monitor. The program you are debugging, which you could name GRAPHIX.EXE, plots points in graphics mode. You want to be able to see the output screen during the debugging session. Finally, you want to be able to start the debugger several times without having to remember all the options, and you want to execute the C start-up code automatically each time. You could create a batch file called GRAFBUG.BAT consisting of the following line:

```
CV /W /B /S /CGmain GRAPHIX
```

The CodeView options are described in more detail in sections 2.5.1–2.5.6.

2.5.1 Starting with a Black-and-White Display

■ Option

/B ¦ –B

The **/B** option forces the CodeView debugger to display in two colors even if you have a CGA. The debugger checks upon start-up to see what kind of display adapter is attached to your computer. If the debugger detects a monochrome adapter, it displays in two colors. If it detects a CGA, it displays in multiple colors.

If you use a two-color monitor with a CGA, you may want to disable color. Monitors that display in only two colors (usually green and black, or amber and black) often attempt to show colors with different cross-hatching patterns, or in gray-scale shades of the display color. In either case, you may find the display easier to read if you use the **/B** option to force black-and-white display. Most two-color monitors still have four color distinctions: background (black), normal text, high-intensity text, and reverse-video text.

■ Example

```
CV /B COUNT COUNT.TXT
```

The example starts the CodeView debugger in black-and-white mode. This is the only mode available if you have a monochrome adapter. The display is usually easier to read in this mode if you have a CGA and a two-color monitor.

2.5.2 Specifying Start-Up Commands

■ Option

/C*commands* | –C*commands*

The **/C** option allows you to specify one or more *commands* that will be executed automatically upon start-up. You can use these options to invoke the debugger from a batch or **MAKE** file. Each command is separated from the previous command by a semicolon.

If one or more of your start-up commands has arguments that require spaces between them, you should enclose the entire option in double quotation marks. Otherwise, the debugger will interpret each argument as a separate CodeView command-line argument rather than as a debugger command argument.

Warning

Any start-up option that uses the less-than (<) or greater-than (>) symbol must be enclosed in double quotation marks even if it does not require spaces. This ensures that the redirection command will be interpreted by the CodeView debugger rather than by MS-DOS.

■ Examples

```
CV /CGmain COUNT.EXE COUNT.TXT

CV "/CS-;N16;G countwords;D buffer L 100" COUNT.EXE COUNT.TXT

CV "/C<INPUT.FIL" COUNT.EXE COUNT.TXT
```

The first example loads the CodeView debugger with `COUNT.EXE` as the executable file and `COUNT.TXT` as the argument. Upon start-up, the debugger executes the C start-up code with the command `Gmain`. Since no space is required between the CodeView command (**G**) and its argument (**main**), the option is not enclosed in double quotation marks.

The second example loads the same file with the same argument, but the command list is more extensive. The debugger starts in assembly mode (`S-`) with a radix of 16 (`N 16`). It executes to procedure `countwords` (`G countwords`), then dumps 100 bytes, starting at the variable `buffer` (`D buffer L 100`). Since several of the commands use spaces, the entire option is enclosed in double quotation marks.

The third example loads the same file and argument, but the start-up command directs the debugger to accept input from the file `INPUT.FIL` rather than from the keyboard. Although the option does not include any spaces, it must be enclosed in double quotation marks so that the less-than symbol will be read by the CodeView debugger rather than by MS-DOS.

2.5.3 Setting the Screen-Exchange Mode

■ Options

/F | –F
/S | –S

The CodeView debugger allows you to move quickly back and forth from the output screen, which contains the output from your program, and the debugging screen, which contains the debugging display. The debugger can handle this screen exchange in two ways: screen flipping or screen swapping. The **/F** option (screen flipping) and the **/S** option (screen swapping) allow you to choose the method from the command line.

If neither method is specified (possible only on non-IBM computers), the Screen Exchange command will not work. No screen exchange is the default for non-IBM computers. Screen flipping is the default for IBM computers with graphics adapters, and screen swapping is the default for IBM computers with monochrome adapters (MAs).

Screen flipping uses the video-display pages of the graphics adapter to store each screen of text. Video-display pages are a special memory buffer reserved for multiple screens of video output. This method is faster and uses less memory than screen swapping. However, screen flipping cannot be used with an MA, or to debug programs that produce graphics or use the video-display pages. In addition, the CodeView debugger's screen flipping only works with IBM and IBM-compatible microcomputers.

Screen swapping has none of the limitations of screen flipping, but is significantly slower and requires more memory. In the screen-swapping method, the CodeView debugger creates a buffer in memory and uses it to store the screen that is not being used. When the user requests the other screen, the debugger swaps the screen in the display buffer for the one in the storage buffer.

When you use screen swapping, the buffer size is 4K if you have an MA, or 16K if you have a CGA or EGA. The amount of memory used by the CodeView debugger is increased by the size of the buffer.

Table 2.1 shows the default exchange mode (swapping or flipping) and the default display mode (sequential or window) for various configurations. Display modes are discussed in Section 2.5.4, "Enabling Window or Sequential Mode."

Table 2.1

Default Exchange and Display Modes

Computer	Display Adapter	Default Modes	Alternate Modes
IBM	CGA or EGA	**/F /W**	**/S** if your program uses video-display pages or graphics; **/T** for sequential mode
IBM compatible	CGA or EGA	**/T**	**/W** for window mode; **/F** for screen flipping with text programs, or **/S** for screen swapping with programs that use video-display pages or graphics
IBM	MA	**/S /W**	**/T** for sequential mode
IBM compatible	MA	**/T**	**/W** for window mode; **/S** for screen swapping
Noncompatible	Any	**/T**	**/S** for screen swapping

If you are not sure if your computer is completely IBM compatible, you can experiment. If the basic input/output system (BIOS) of your computer is not compatible enough, the CodeView debugger may not work with the **/F** option.

If you specify the **/F** option with an MA, the debugger will ignore the option and use screen swapping. If you try to use screen flipping to debug a program that produces graphics or uses the video-display pages, you may get unexpected results and have to start over with the **/S** option.

■ Examples

```
CV /F COUNT COUNT.TXT

CV /S GRAFIX
```

The first example starts the CodeView debugger with screen flipping. You might use this command line if you have an IBM-compatible computer, and you want to override the default screen-exchange mode in order to use less memory and switch screens more quickly. The option would not be necessary on an IBM computer, since screen flipping is the default.

The second example starts the debugger with screen swapping. You might use this command line if your program uses graphics mode.

2.5.4 Enabling Window or Sequential Mode

■ Options

/T | –T
/W | –W

The CodeView debugger can operate in window mode or in sequential mode. Window mode displays up to four windows, enabling you to see different aspects of the debugging session program simultaneously. You can also use a mouse in window mode. Window mode requires an IBM or IBM-compatible microcomputer.

Sequential mode works with any computer, and is useful with redirection commands. Debugging information is displayed sequentially on the screen.

The behavior of each mode is discussed in detail in Chapter 3, "The CodeView Display." Refer back to Table 2.1 for the default and alternate modes for your computer. If you are not sure if your computer is completely IBM compatible, you can experiment with the options. If the BIOS of your computer is not compatible enough, you may not be able to use window mode (the **/W** option).

Note

Although window mode is more convenient, any debugging operation that can be done in window mode can also be done in sequential mode.

■ Examples

```
CV /W SIEVE

CV /T SIEVE
```

The first example starts the CodeView debugger in window mode. You will probably want to use the **/W** option if you have an IBM-compatible computer, since the default sequential mode is less convenient for most debugging tasks.

The second example starts the debugger in sequential mode. You might want to use this option if you have an IBM computer and you have a specific reason for using sequential mode. For instance, sequential mode usually works better if you are redirecting your debugging output to a remote terminal.

2.5.5 Turning Off the Mouse

■ Option

/M | –M

If you have a mouse installed on your system, you can tell the CodeView debugger to ignore it, using the **/M** option. You may need to use this option if you are debugging a program that uses the mouse and your mouse is not a Microsoft Mouse. This is due to a conflict between the program's use of the mouse and the debugger's use of it. If you use the **/M** option, the program you are debugging can still use the mouse, but the CodeView debugger cannot.

Important

The same conflict between program and debugger applies if you have a version of the Microsoft Mouse prior to 5.02. The latest version of the mouse driver program (**MOUSE.SYS**) is included on the Microsoft C Compiler distribution disk. You should replace your old mouse driver program with this updated version. You will then be able to use the mouse both with the CodeView debugger and the program you are debugging. See your Microsoft Mouse user's guide for information on installing **MOUSE.SYS**. This file will not work with pointing devices from other manufacturers.

2.5.6 Using the Enhanced Graphics Adapter's 43-Line Mode

■ Option

/43 | –43

If you have an EGA, you can use the **/43** option to enable a 43-line-by-80-column text mode. You cannot not use this mode if you have a CGA or an MA. The CodeView debugger will ignore the option if it does not detect an EGA.

The EGA's 43-line mode performs identically to the normal 25-line-by-80-column mode used by default on the EGA, CGA, and MA. The advantage of the 43-line mode is that more text fits on the CodeView display; the disadvantage is that the text is smaller and harder to read. If you have an EGA, you can experiment to see which size you prefer.

■ Example

```
CV /43 COUNT COUNT.TXT
```

This example starts the CodeView debugger in 43-line mode if you have an EGA video adapter and monitor. The option will be ignored if you do not have the hardware to support it.

2.6 Using the CodeView Debugger with the Macro Assembler

You can use the CodeView debugger with files developed with the Microsoft (or IBM) Macro Assembler. Since the Microsoft Macro Assembler (versions 1.0 through 4.0) does not write line numbers to object files, some of the CodeView debugger's features will not be used when you debug programs developed with the assembler.

The debugger can be used on either **.EXE** or **.COM** files, but you can only view symbolic information in **.EXE** files. The procedure for assembling and debugging **.EXE** files is summarized below:

1. In your source file, declare public any symbols, such as labels and variables, that you want to reference in the debugger. If the file is small, you may want to declare all symbols public.
2. Assemble as normal. No special options are required, and all assembly options are allowed.
3. Link with the linker provided with Version 4.0 of the Microsoft C Compiler. Do not use the linker provided with versions 4.0 and earlier of the Macro Assembler. Use the **/CODEVIEW** option when linking.
4. Debug in assembly mode (this is the start-up default if the debugger doesn't find line-number information). You cannot use source mode for debugging, but you can load the source file into the display window and view it in source mode. You may find this convenient for referring to macros and comments. Any labels or variables that you declared public in the source file can be displayed and referenced by name instead of by address.

You can also use this procedure to debug C library routines or assembly-language modules called by your C program.

Chapter 3
The CodeView Display

3.1 Introduction 35

3.2 Using Window Mode 36

3.2.1 Executing Window Commands with the Keyboard 38

3.2.1.1 Moving the Cursor with Keyboard Commands 38

3.2.1.2 Changing the Screen with Keyboard Commands 40

3.2.1.3 Controlling Program Execution with Keyboard Commands 41

3.2.1.4 Selecting from Menus with the Keyboard 42

3.2.2 Executing Window Commands with the Mouse 44

3.2.2.1 Changing the Screen with the Mouse 44

3.2.2.2 Controlling Program Execution with the Mouse 45

3.2.2.3 Selecting from Menus with the Mouse 47

3.2.3 Using Menu Selections 49

3.2.3.1 Using the File Menu 49

3.2.3.2 Using the Search Menu 51

3.2.3.3 Using the View Menu 53

3.2.3.4 Using the Run Menu 54

3.2.3.5 Using the Watch Menu 55

3.2.3.6 Using the Options Menu 57

3.2.3.7 Using the Calls Menu 60

3.2.4 Using the Help System 61

3.3 Using Sequential Mode 63

3.1 Introduction

The CodeView screen display can appear in two different modes: window and sequential. Both modes provide a useful debugging environment, but the window mode is the more powerful and convenient of the two.

Most users will prefer to use window mode, if they have the hardware to support it. In window mode, popup menus, function keys, and mouse support offer fast access to the most common commands. Different aspects of the program and debugging environment can be seen in different windows simultaneously. Window mode is described in Section 3.2.

Sequential mode should be familiar to most programmers. It is similar to the display mode of the CodeView debugger's predecessors, the Microsoft Symbolic Debugging Utility (**SYMDEB**) and the MS-DOS **DEBUG** utility. This mode is required if you do not have an IBM-compatible computer, and it is sometimes useful when redirecting command input or output. Sequential mode is described in Section 3.3.

3.2 Using Window Mode

Figure 3.1 shows the CodeView window-mode display with all windows open.

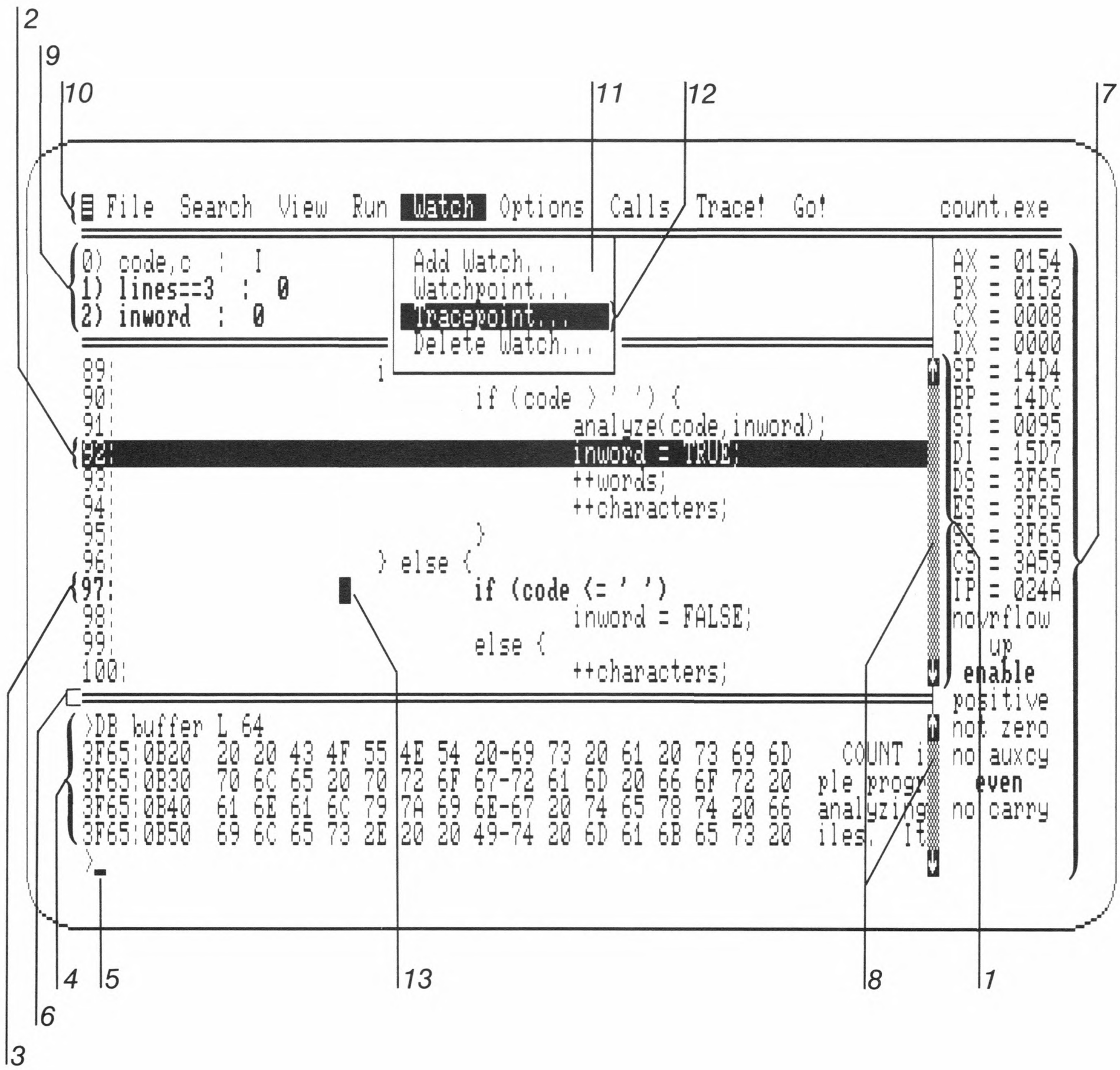


Figure 3.1 Elements of the CodeView Debugging Screen

The elements of the CodeView display marked in Figure 3.1 are explained below:

1. The display window shows the program being debugged. It can contain source code (as in the example), assembly-language instructions, or any specified text file.
2. The current location line (the next line the program will execute) is displayed in reverse video or in a different color. This line may not always be visible, since you can scroll to earlier or later parts of the program.
3. Lines containing previously set breakpoints are shown in high-intensity text.
4. The dialog window is where you enter dialog commands. These are the commands with optional arguments that you can enter at the CodeView prompt (>). You can scroll up or down in this window to view previous dialog commands and command output.
5. The cursor is a thin blinking line that shows the location at which you can enter commands from the keyboard. You can move the cursor up and down, and put it in either the dialog or display window.
6. The display/dialog separator line divides the dialog window from the display window. You can move this line up or down to change the relative size of the two windows.
7. The register window shows the current status of the registers and flags. This is an optional window that can be opened or closed with one keystroke.
8. The scroll bars are the vertical bars on the right side of the screen. Each scroll bar has an up arrow and a down arrow that you can use to scroll through the display with a mouse.
9. The watch window is an optional window that shows the current status of specified variables or expressions. It appears automatically whenever you create watch statements. See Chapter 8, "Managing Watch Statements."
10. The menu bar shows titles of menus and commands that you can activate with the keyboard or the mouse. Titles followed by an exclamation point represent commands; other titles are menus.
11. Menus can be opened by specifying the appropriate title on the menu bar. On the sample screen, the Watch menu has been opened.

12. The menu "highlight" is a reverse-video or colored strip indicating the current selection in a menu. You can move the highlight up or down to change the current selection.
13. The mouse pointer indicates the current position of the mouse. It is shown only if you have a mouse installed on your system.
14. Dialog boxes (not shown) appear in the center of the screen when you choose a menu selection that requires a response. The box prompts you for a response and disappears when you enter your answer.
15. Message boxes (not shown) appear in the center of the screen to display errors or other messages.

The screen elements are described in more detail in the rest of this chapter.

3.2.1 Executing Window Commands with the Keyboard

CodeView accepts two kinds of commands: window commands and dialog commands. Dialog commands are entered as command lines following the CodeView prompt (>) in sequential mode. They are discussed in Chapter 4, "Using Dialog Commands."

The most common CodeView debugging commands and all the commands for managing the CodeView display are available with window commands. Window commands are one-keystroke commands that can be entered with function keys, CONTROL-key combinations, ALT-key combinations, or the direction keys on the numeric keypad.

Most window commands can also be entered with a mouse, as described in Section 3.2.2. The window commands available from the keyboard are described by category in the following sections. For a table of commands by name, see Appendix A, "Command and Mode Summary."

3.2.1.1 Moving the Cursor with Keyboard Commands

The following keys move the cursor or scroll text up or down in the display or dialog window:

Key	Function
F6	Moves the cursor between the display and dialog windows. If the cursor is in the dialog window when you press F6, it will move to its previous position in the display window. If the cursor is in the display window, it will move to its previous position in the dialog window.
CONTROL-U	Moves the display/dialog separator line up one line. This decreases the size of the display window and increases the size of the dialog window. If the cursor is in the dialog window, you can remove the display window entirely by moving the separator line to the top of the window.
CONTROL-D	Moves the display/dialog separator line down one line. This increases the size of the display window and decreases the size of the dialog window. If the cursor is in the display window, you can remove the dialog window entirely by moving the separator line to the bottom of the screen.
UP ARROW	Moves the cursor up one line in either the display or dialog window.
DOWN ARROW	Moves the cursor down one line in either the display or dialog window.
PGUP	Scrolls up one page. If the cursor is in the display window, the source lines or assembly-language instructions scroll up. If the cursor is in the dialog window, the buffer of commands entered during the session scrolls up. The cursor remains at its current position in the window. The length of a page is the current number of lines in the window.
PGDN	Scrolls down one page. If the cursor is in the display window, the source lines or assembly-language instructions scroll down. If the cursor is in the dialog window, the buffer of commands entered during the session scrolls down. The cursor remains at its current position in the window. The length of a page is the current number of lines in the window.
HOME	Scrolls to the top of the file or command buffer. If the cursor is in the display window, the text scrolls to the start of the source file or program instructions. If the cursor is in the dialog window, the commands scroll

	to the top of the command buffer. The top of the command buffer may be blank if you have not yet entered enough commands to fill the buffer. The cursor remains at its current position in the window.
END	Scrolls to the bottom of the file or command buffer. If the cursor is in the display window, the text scrolls to the end of the source file or program instructions. If the cursor is in the dialog window, the commands scroll to the bottom of the command buffer and the cursor moves to the CodeView prompt (>) at the end of the buffer.

3.2.1.2 Changing the Screen with Keyboard Commands

The following keys change the screen, or switch to a different screen:

Key	Function
F1	Displays initial on-line-help screen. The help system is discussed in Section 3.2.4. You can also get to the initial help screen by selecting Help from the View menu, as described in Section 3.2.3.3.
F2	Toggles the register window. The window disappears if present, or appears if absent. You can also toggle the register window with the Register selection from the Options menu, as described in Section 3.2.3.6.
F3	Switches between source and assembly modes. Source mode shows source code in the display window, while assembly mode shows assembly-language instructions. You can also change modes with the Source and Assembly selections from the View menu, as described in Section 3.2.3.3.
F4	Switches to the output screen. The output screen shows the output, if any, from your program. Press any key to return to the CodeView screen. You can also change to the output screen with the Output selection from the View menu, as described in Section 3.2.3.3.

3.2.1.3 Controlling Program Execution with Keyboard Commands

The following keys set and clear breakpoints, trace through your program, or execute to a breakpoint:

Key	Function
F5	Executes to the next breakpoint, or to the end of the program if no breakpoint is encountered. This keyboard command corresponds to the Go dialog command when it is given without a destination breakpoint argument.
F7	Sets a temporary breakpoint on the line with the cursor, and executes to that line (or to a previously set breakpoint or the end of the program if either is encountered before the temporary breakpoint). In source mode, if the line does not correspond to code (for example, data declaration or comment lines), the CodeView debugger sounds a warning and ignores the command. This window command corresponds to the Go dialog command when it is given with a destination breakpoint.
F8	Executes a Trace command. The CodeView debugger executes the next source line in source mode, or the next instruction in assembly mode. If the source line or instruction contains a function, procedure, or interrupt call, the debugger starts tracing through the call (enters the call and is ready to execute the first source line or instruction). This command will not trace into MS-DOS function calls (interrupt 0x21).
F9	Sets or clears a breakpoint on the line with the cursor. If the line does not currently have a breakpoint, one is set on that line. If the line already has a breakpoint, the breakpoint is cleared. If the cursor is in the dialog window, the CodeView debugger sounds a warning and ignores the command. This window command corresponds to the Breakpoint Set and Breakpoint Clear dialog commands.
F10	Executes the Program Step command. The CodeView debugger executes the next source line in source mode, or the next instruction in assembly mode. If the source line or instruction contains a function, procedure, or interrupt call, the debugger steps over the

entire call (executes it to the return) and is ready to execute the line or instruction after the call.

Important

You can usually interrupt program execution by pressing CONTROL-BREAK or CONTROL-C. This can be used to exit endless loops, or it can interrupt loops that are slowed by the Watchpoint or Tracepoint commands (see Chapter 8, "Managing Watch Statements"). CONTROL-BREAK or CONTROL-C may not work if your program has a special use for one or both of these key combinations. If you have an IBM Personal Computer AT (or a compatible computer), you can use the SYSTEM-REQUEST key to interrupt execution regardless of your program's use of CONTROL-BREAK and CONTROL-C.

3.2.1.4 Selecting from Menus with the Keyboard

The CodeView debugger has seven popup menus. This section discusses how to make selections from menus. The effects of the selections are discussed in Section 3.2.3.

The menu bar at the top of the screen has nine titles: File, Search, View, Run, Watch, Options, Calls, Trace!, and Go!. The first seven titles are menus, and the last two are commands. The Trace! and Go! titles are provided primarily for mouse users, though you can activate them by pressing ALT-T or ALT-G and then pressing the ENTER key. The exclamation point is a convention used to indicate that a title represents a command rather than a menu. The same commands are more easily accessible with the F5, F8, and F10 keys.

The steps for opening a menu and making a selection are described below:

1. To open a menu, press the ALT key and the first letter of the menu title. For example, press ALT-S to open the Search menu. The menu title is highlighted and a menu box listing the selections pops up below the title.

2. There are two ways to make a selection from an open menu:

   a. Press the DOWN ARROW key on the numeric keypad to move down the menu. The highlight will follow your movement. When the item you want is highlighted, press the ENTER key to

execute the command. For example, press the DOWN ARROW once to select Find from the Search menu.

You can also press the UP ARROW key to move up the menu. If you move off the top or bottom of the menu, the highlight wraps around to the other end of the menu.

b. While holding down the ALT key, press the first letter of the item you want to select. You do not have to press the ENTER key with this method. For example, press ALT-F to select Find from the Search menu. This selection method does not work with the Calls menu.

3. One of three things will happen at this point:

   a. For most menu selections, the choice is executed immediately.

   b. The items on the Options menu have small double arrows next to them if the option is on, and no arrows if the option is off. Choosing the item toggles the option. The status of the arrows will be reversed the next time you open the menu.

   c. Some items require a response. In this case, there is another step in the menu-selection process.

4. If the item you select requires a response, a dialog box opens when you select a menu item. Type your response to the prompt in the box and press the ENTER key. For example, the Find dialog box asks you to enter a regular expression (see Section 3.2.3.2, "Using the Search Menu," or Chapter 10, "Modifying Code or Data," for an explanation of regular expressions).

   If your response is valid, the command will be executed. If you enter an invalid response, a message box will appear, telling you the problem and asking you to press a key. Press any key to make the message box disappear.

At any point during the process of selecting a menu item, you can press the ESCAPE key to cancel the menu. While a menu is open, you can press the LEFT ARROW or RIGHT ARROW key to move from one menu to an adjacent menu, or to one of the command titles on the menu bar.

3.2.2 Executing Window Commands with the Mouse

The CodeView debugger is designed to work with the Microsoft Mouse (it also works with some compatible pointing devices). By moving the mouse on a flat surface, you can move the mouse pointer in a corresponding direction on the screen. The following terms refer to the way you select items or execute commands with the mouse:

Term	Definition
Point	To move the mouse until the mouse pointer rests on the item you want to select.
Click	To quickly press and release a mouse button while pointing at an item you want to select.
Drag	To press a mouse button while on a selected item, then hold the button down while moving the mouse. The item moves in the direction of the mouse movement. When the item you are moving is where you want it, release the button; the item will stay at that point.

The CodeView debugger uses two mouse buttons. The terms "click right," "click left," "click both," and "click either" are sometimes used to designate which buttons to use. When dragging, either button can be used.

3.2.2.1 Changing the Screen with the Mouse

You can change various aspects of the screen display by pointing to one of the following elements and then either clicking or dragging:

Item	Action
Double line separating display and dialog windows	Drag the separator line up to increase the size of the dialog window while decreasing the size of the display window, or drag the line down to increase the size of the display window while decreasing the size of the dialog window. You can eliminate either window completely by dragging the line all the way up or down (providing the cursor is not in the window you want to eliminate).

UP ARROW or DOWN ARROW on the scroll bar

Point and click on one of the four arrows on the scroll bars to scroll up or down. If you are in the display window, source code will scroll up or down. If you are in the dialog window, the buffer containing dialog commands entered during the session will scroll up or down. The distance moved is determined by which buttons you click as follows:

Button	Action
Click left	Scroll up or down, one line at a time
Click right	Scroll up or down, one page at a time; the length of a page is the current size of the window
Click both	Scroll to the top or bottom of the file or command buffer

Some menu selections also change the screen display. See Section 3.2.3 for a description of the menu selections.

3.2.2.2 Controlling Program Execution with the Mouse

By clicking on the following mouse items, you can set and clear breakpoints, trace through your program, execute to a breakpoint, or change the flag bits:

Item	Action
Source line or instruction	Point and click on a source line in source mode or on an instruction in assembly mode to take one of the following actions:

Button	Action
Click left	If the line under the mouse cursor does not have a breakpoint, one is set there. If the line already has a breakpoint, the breakpoint is removed. Lines with breakpoints are shown in high-intensity text.

Click right	A temporary breakpoint is set on the line and the CodeView debugger executes until it reaches the line (or until it reaches a previously set breakpoint or the end of the program if either is encountered before the temporary breakpoint).

If you click on a line that does not correspond to code (for example, a declaration or comment), the CodeView debugger will sound a warning and ignore the command.

Trace! on menu bar

Point and click to trace the next instruction. The kind of trace is determined as follows by the button clicked:

Button	Action
Click left	The Trace command is executed. The CodeView debugger executes the next source line in source mode, or the next instruction in assembly mode. If the source line or instruction contains a function, procedure, or interrupt call, the debugger starts tracing through the call (it enters the call and is ready to execute the first source line or instruction). This command will not trace into MS-DOS function calls (interrupt 0x21).
Click right	The Program Step command is executed. The debugger executes the next source line in source mode, or the next instruction in assembly mode. If the source line or instruction contains a function, procedure, or interrupt call, CodeView steps over the entire call (it executes the call to the return) and is ready to execute the line or instruction after the call.

	These two commands are only different if the current location is the start of a procedure, interrupt, or function call.
Go! on menu bar	Point and click either button to execute to the next breakpoint, or to the end of the program if no breakpoints are encountered.
Flag in register window	Point to a flag name and click either button to reverse the flag. If the flag bit is set, it will be cleared; if the flag bit is cleared, it will be set. The flag name is changed on the screen to match the new status. If you are using color mode, the color of the flag mnemonic will also change. This command can only be used when the register window is open. Use the command with caution, since changing flag bits can change program execution at the lowest level.

Important

You can usually interrupt program execution by pressing CONTROL-BREAK or CONTROL-C. See the note in Section 3.2.1.3 for more information.

3.2.2.3 Selecting from Menus with the Mouse

The CodeView debugger has seven popup menus. This section discusses how to make selections from these menus. The effect of each selection is discussed in Section 3.2.3.

The menu bar at the top of the screen has nine titles: File, Search, View, Run, Watch, Options, Calls, Trace!, and Go!. The first seven titles are menus and the last two are commands that you can execute by clicking with the mouse. The steps for opening a menu and making a selection are described below:

1. To open a menu, point to the title of the menu you want to select. For example, move the pointer onto File on the menu bar if you want to open the File menu.

2. With the mouse pointer on the title, press and hold down either mouse button. The selected title is highlighted and a menu box with a list of selections pops up below the title. For example, if you point to Search and press a button, the Search menu pops up.

3. With the button held down, move the mouse down. The highlight follows the mouse movement. You can move the highlight up or down in the menu box. For example, to select Find from the Search menu, move the highlight down the menu to Find.

   If you move off the box, the highlight will disappear. However, as long as you do not release the button, you can move the pointer back onto the menu to make the highlight reappear.

4. When the selection you want is highlighted, release the mouse button. For example, release the button with the highlight on Find.

   When you release the button, the menu selection is executed. One of three things will happen:

   a. For most menu selections, the choice is executed immediately.

   b. The items on the Options menu have small double arrows next to them if the option is on, or no arrows if the option is off. Choosing the item toggles the option. The status of the arrows will appear reversed the next time you open the menu.

   c. Some items require a response. In this case, there is another step in the menu-selection process.

5. If the item you select requires a response, a dialog box with a prompt appears. Type your response and press the ENTER key or a mouse button. For example, if you selected Find, the prompt will ask you to enter a regular expression (see Section 3.2.3.2, "Using the Search Menu," or Appendix B, "Regular Expressions," for an explanation of regular expressions).

   If your response is valid, the command will be executed. If you enter an invalid response in the dialog box, a message box will appear telling you the problem and asking you to press a key. Press any key or click a mouse button to make the message box disappear.

There are several shortcuts you can take when selecting menu items with the mouse. If you change your mind and decide not to select an item from a menu, just move off the menu and release the mouse button; the menu will disappear. You can move from one menu to another by dragging the pointer directly from any point on the current menu to the title of the new menu.

3.2.3 Using Menu Selections

This section describes the selections on each of the CodeView menus. These selections can be made with the keyboard, as described in Section 3.2.1, or with the mouse, as described in Section 3.2.2.

3.2.3.1 Using the File Menu

The File menu includes selections for working on the current source or program file. The File menu is shown in Figure 3.2, and the selections are explained below:

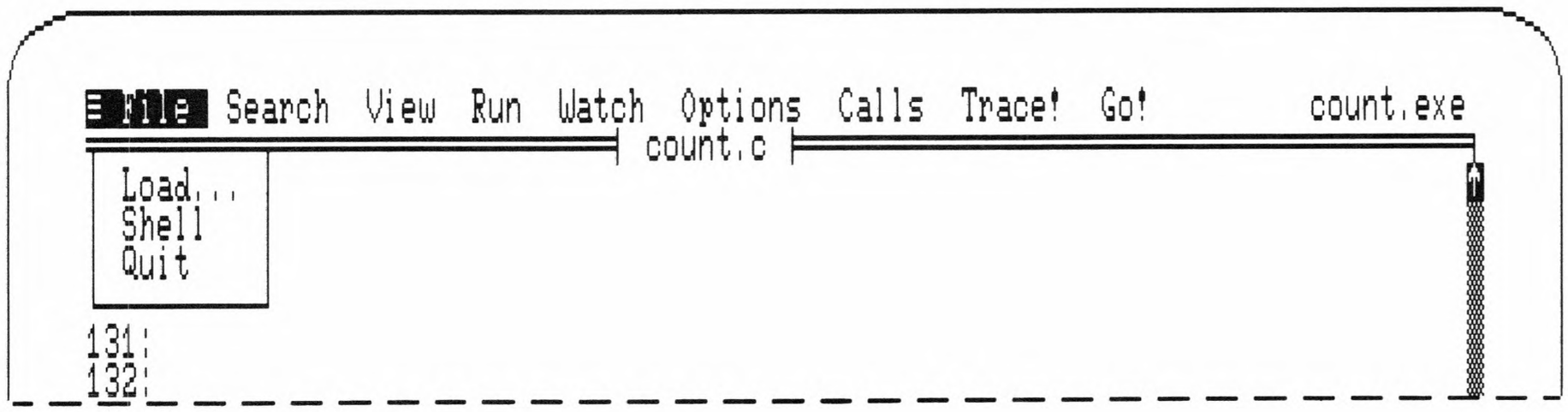


Figure 3.2 The File Menu

Selection	Action
Load...	Opens a new file. When you make this selection, a dialog box appears asking for the name of the new file you want to open. Type the name of a source file, an include file, or any other text file. The text of the new file replaces the current contents of the display window (if you are in assembly mode, the CodeView debugger will switch to source mode). When you finish viewing the file, you can reopen the original source file. The current location and breakpoints will still be marked when you return to the source file. You do not need to open a new file to see source files for a different module of your program. The CodeView debugger automatically switches to the source file of the other module when program execution enters the other module. While switching source files is never necessary, it may be desirable if you want to

set breakpoints or execute to a line in another module.

Note

If the debugger cannot find the source file when it switches modules, a dialog box appears asking for a file specification for the source file. You can either enter a new file specification if the file is in another directory, or press the ENTER key if no source file exists. If you press the ENTER key, the module can only be debugged in assembly mode.

Shell

Exits to an MS-DOS shell. This brings up the MS-DOS screen, where you can execute MS-DOS commands or executable files. To return to the CodeView debugger, type `exit` at the MS-DOS command prompt. The CodeView screen reappears with the same status it had when you left it.

The Shell command works by saving the current processes in memory and then executing a second copy of **COMMAND.COM**. This requires a significant amount of free memory (more than 200K), since the debugger, **COMMAND.COM**, symbol tables, and the debugged program must all be saved in memory. If you do not have enough memory to execute the Shell command, an error message appears. Even if you have enough memory to execute the shell, you may not have enough memory left to execute large programs from the shell.

The Shell command will not work unless you have executed the C start-up code. You can do this after starting the debugger, or after restarting your program, by executing to any point within the program. For example, enter the dialog command `G main`.

Quit

Terminates the CodeView debugger and returns to MS-DOS.

3.2.3.2 Using the Search Menu

The Search menu includes selections for searching through text files for text strings and for searching executable code for labels. The Search menu is show in Figure 3.3 and the selections are explained below:

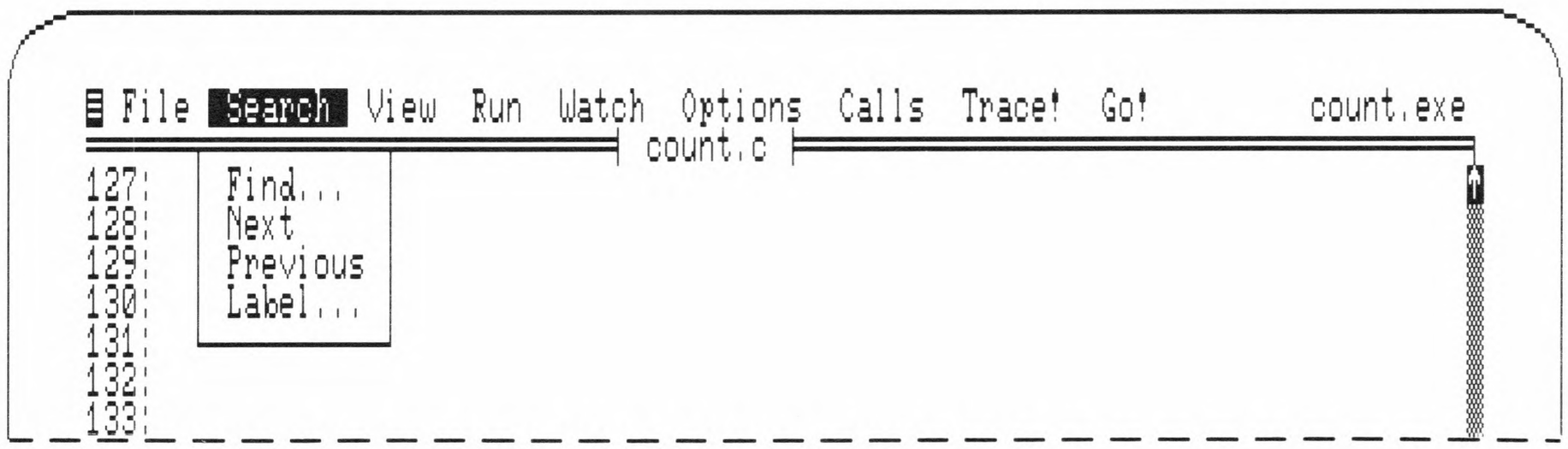


Figure 3.3 The Search Menu

Selection	Action
Find...	Searches the current source file or other text file for a specified regular expression. When you make this selection, a dialog box opens, asking you to enter a regular expression. Type the expression you want to search for and press the ENTER key. CodeView starts at the current or most recent cursor position in the display window and searches for the expression. If your entry is found, the cursor moves to the first source line containing the expression. If the entry is not found, a message box opens, telling you the problem and asking you to press a key (you can also click a mouse button) to continue. If you are in assembly mode, the debugger automatically switches to source mode when the expression is found. Regular expressions are a method of specifying variable text strings. This method comes from the XENIX® and UNIX™ operating systems, and is similar to the MS-DOS method of using wild cards in file names. Regular expressions are explained in detail in Appendix B. You can use the Search selections without understanding regular expressions. Since text strings are

the simplest form of regular expressions, you can simply enter a string of characters as the expression you want to find. For example, you could enter `count` if you wanted to search for the word "count."

The following characters have a special meaning in regular expressions: backslash (\), asterisk (*), left bracket ([), period (.), dollar sign ($), and caret (^). In order to find strings containing these characters, you must precede the characters with a backslash; this cancels their special meanings.

For example, you would use `\\n` to find `\n` or use `buffer\[count]` to find `buffer[count]`. The period in some member-selection expressions and the caret in the XOR operator and the XOR assignment operator must also be preceded by a backslash.

Next — Searches for the next match of the current regular expression. This selection is only meaningful after you have used the Search command to specify the current regular expression. If the CodeView debugger searches to the end of the file without finding another match for the expression, it wraps around and starts searching at the beginning of the file.

Previous — Searches for the previous match of the current regular expression. This selection is only meaningful after you have used the Search command to specify the current regular expression. If the debugger searches to the beginning of the file without finding another match for the expression, it wraps around and starts searching at the end of the file.

Label... — Searches the executable code for a label. A label can be a function name or an assembly-language label. If the label is found, the cursor moves to the source line or instruction containing the label. The debugger will switch to assembly mode, if necessary, to show a label in a library routine or an assembly-language module.

3.2.3.3 Using the View Menu

The View menu includes selections for switching between source and assembly modes, and for switching among the debugging screen, the output screen, and the help screen. The corresponding function keys for menu selection are shown on the right side of the menu when appropriate. The View menu is shown in Figure 3.4, and the selections are explained below:

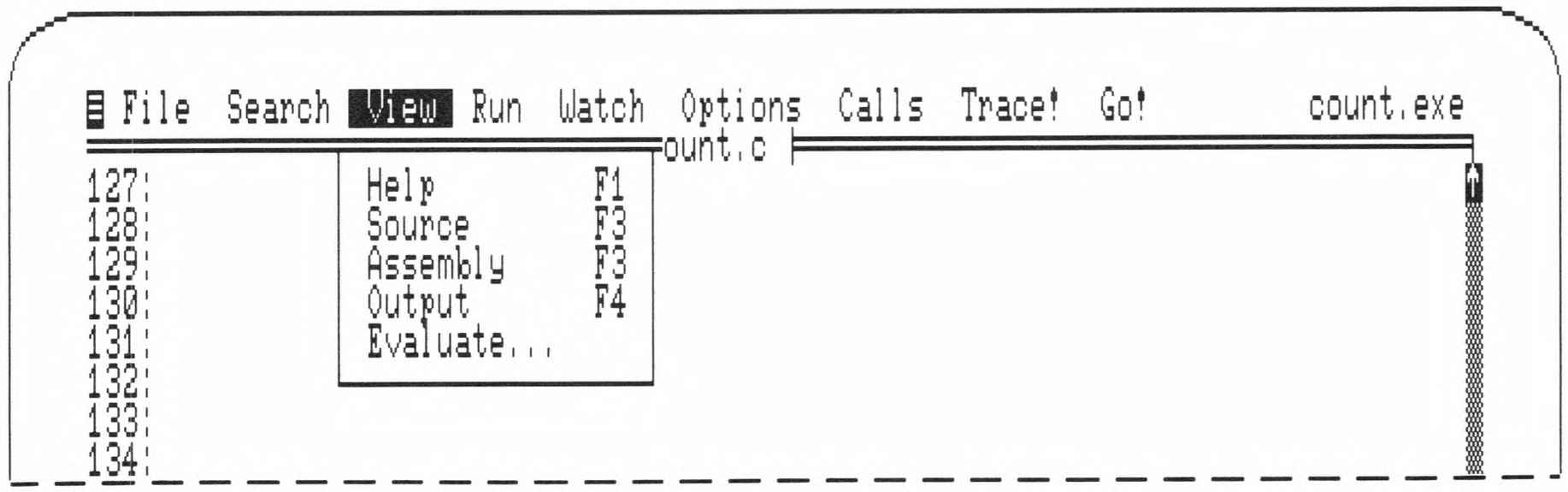


Figure 3.4 The View Menu

Selection	Action
Help	Opens the initial help menu. Section 3.2.4 tells how to move through the on-line-help system and return to the debugging screen.
Source	Changes from assembly mode (showing assembly-language instructions in the display window) to source mode (showing source lines). If you select this mode when you are already in source mode, the selection will be ignored.
Assembly	Changes from source mode (showing source lines in the display window) to assembly mode (showing assembly-language instructions). If you select this mode when you are already in assembly mode, your selection will be ignored.
Output	Replaces the CodeView screen with the output screen. The output screen shows the current output of your program. If the program has not taken over the entire screen, previous MS-DOS commands may still be visible on the output screen. To return to the

CodeView screen, press any key or click a mouse button. The output screen is only for viewing; you cannot change it.

Evaluate... — Evaluates a C expression. A dialog box opens and asks for the expression to be evaluated. Enter a C expression made up of identifiers and the C operators recognized by the CodeView debugger (see Chapter 4, "Using Dialog Commands"). The value of the expression appears at the CodeView prompt in the dialog window. This selection is similar to the Display Expression dialog command.

You can specify the format in which the value will be displayed. Type the expression, followed by a comma and a **printf** type specifier. If you do not give a type specfier, the debugger displays the value in a default format. See Chapter 6, "Examining Data and Expressions," or Appendix A, "Command and Mode Summary," for more information about type specifiers and the default format.

3.2.3.4 Using the Run Menu

The Run menu includes selections for running your program. The Run menu is shown in Figure 3.5, and the selections are explained below:

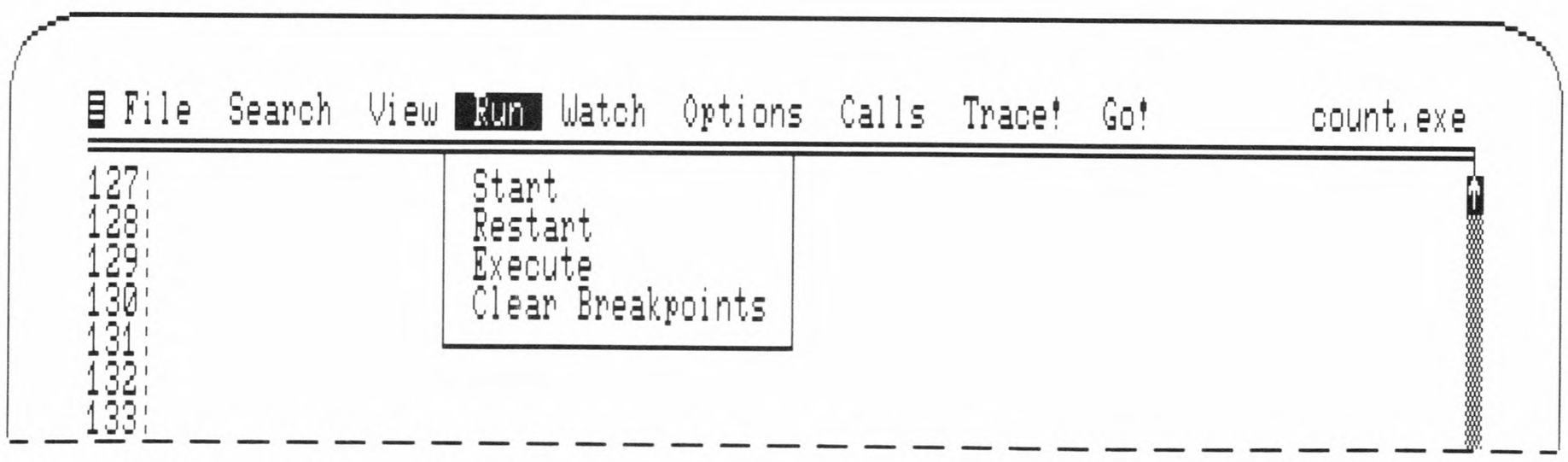


Figure 3.5 The Run Menu

Selection	Action
Start	Starts the program from the beginning and runs it. Any previously set breakpoints or watch statements

will still be in effect. The CodeView debugger will run your program from the beginning to the first breakpoint, or to the end of the program if no breakpoint is encountered. This has the same effect as selecting Restart (see the next selection) and then entering the Go command.

Restart	Restarts the current program, but does not begin executing it. You can debug the program again from the beginning. Any previously set breakpoints or watch statements will still be in effect.
Execute	Executes in slow motion from the current instruction. This is the same as the Execute dialog command (**E**). To stop animated execution, press any key or a mouse button.
Clear Breakpoints	Clears all breakpoints. This selection may be convenient after selecting Restart if you don't want to use previously set breakpoints. Note that watch statements are not cleared by this command.

3.2.3.5 Using the Watch Menu

The Watch menu includes selections for managing the watch window. Selections on this menu are also available with dialog commands. The Watch menu is shown in Figure 3.6, and the selections are explained below:

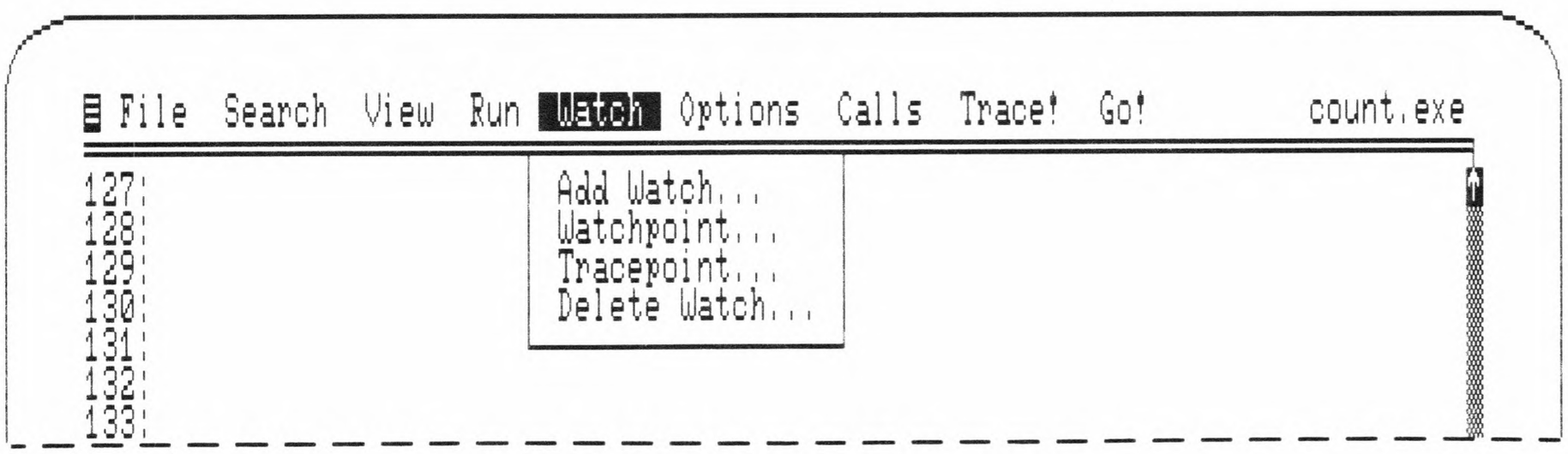


Figure 3.6 The Watch Menu

Selection	Action
Add Watch...	Adds a watch-expression statement to the watch window. A dialog window opens, asking for the C expression whose value you want to see displayed in the watch window. Type the expression and press the ENTER key or press a mouse button. The statement appears in the watch window in normal text. You cannot specify a memory range to be displayed with the Add Watch selection as you can with the Watch dialog command. You can specify the format in which the value will be displayed. Type the expression, followed by a comma and a **printf** type specifier. If you do not give a type specifier, the CodeView debugger displays the value in a default format. See Chapter 6, "Examining Data and Expressions," or Appendix A, "Command and Mode Summary," for more information about type specifiers and the default format. See Section 8.2, "Setting Watch-Expression or Watch-Memory Statements," for more information about the Watch command.
Watchpoint...	Adds a watchpoint statement to the watch window. A dialog window opens, asking for the C expression whose value you want to test. The watchpoint statement appears in the watch window in high-intensity text when you enter the expression. A watchpoint is a conditional breakpoint that causes execution to stop when the expression becomes nonzero (true). See Section 8.3, "Setting Watchpoints," for more information.
Tracepoint...	Adds a tracepoint statement to the watch window. A dialog window opens, asking for the C expression or memory range whose value you want to test. The tracepoint statement appears in the watch window in high-intensity text when you enter the expression. A tracepoint is a conditional breakpoint that causes execution to stop when the value of a given expression changes. You cannot specify a memory range to be tested with the Tracepoint selection as you can with the Tracepoint dialog command.

When setting a tracepoint expression, you can specify the format in which the value will be displayed. Type the expression followed by a comma and a type specifier. If you do not give a type specifier, the CodeView debugger displays the value in a default format. See Chapter 6, "Examining Data and Expressions," for more information about type specifiers and the default format. See Section 8.4, "Setting Tracepoints," for more information about tracepoints.

Delete Watch... Deletes a watch statement from the watch window. A dialog window opens, showing the current watch statements. If you are using a mouse, move the pointer to the statement you want to delete and click either button. If you are using the keyboard, press the UP ARROW or DOWN ARROW key to move the highlight to the statement you want to delete, then press the ENTER key.

3.2.3.6 Using the Options Menu

The Options menu allows you to set options that affect various aspects of the behavior of the CodeView debugger. The Options menu is shown in Figure 3.7, and the selections are explained below:

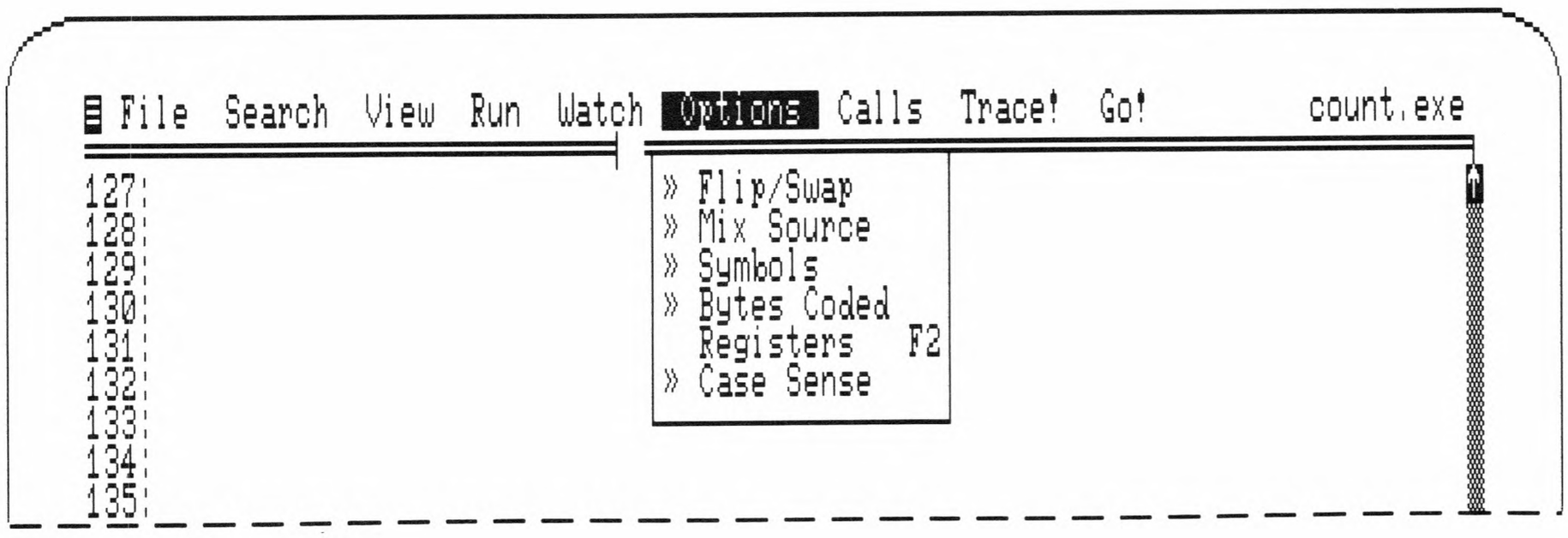


Figure 3.7 The Options Menu

Three of the options control the way assembly-language instructions are displayed in assembly mode. The default setting for assembly mode shows source lines, if available, with the corresponding assembly-language

instructions; the bytes for each instruction are shown as well as the instruction mnemonics; and symbols are used to show variables and labels. The default assembly display for one source line is shown below:

```
27:                 name = gets(namebuf);
32AF:003E 8D46DE            LEA       AX,Word Ptr [namebuf]
32AF:0041 50                PUSH      AX
32AF:0042 E89C03            CALL      _gets (03E1)
32AF:0045 83C402            ADD       SP,02
32AF:0048 8946DA            MOV       Word Ptr [name],AX
```

Compare this display with the variations shown in the descriptions for each assembly option.

Selections on the Options menu have small double arrows to the left of the selection name when the option is on. The status of the option (and the presence of the double arrows) is reversed each time you select the option. By default, all the options except Registers are on when you start the CodeView debugger.

The selections from the Options menu are discussed below:

Selection	Action
Flip/Swap	When on (the default), screen swapping or screen flipping (whichever the debugger was started with) is active; when off, swapping or flipping is disabled. Turning off swapping or flipping makes the screen scroll more smoothly. You will not see the program flip or swap each time you execute part of the program. This option has no effect if neither swapping nor flipping was selected during start-up.

Warning

Make sure that flipping or swapping is on any time your program writes to the screen. If swapping and flipping are off, your program will write to the dialog window. The CodeView debugger will detect that the screen has changed and will redraw the screen, thus destroying the program output.

Mix Source

When on (the default), source lines are shown with assembly language instructions; when off, instructions are shown without source lines. This option only affects assembly mode. The sample default display at the beginning of this section shows the appearance of the screen when the option is on. The following display shows the appearance of the same code when the option is off:

```
32AF:003E 8D46DE        LEA     AX,Word Ptr [namebuf]
32AF:0041 50            PUSH    AX
32AF:0042 E89C03        CALL    _gets (03E1)
32AF:0045 83C402        ADD     SP,02
32AF:0048 8946DA        MOV     Word Ptr [name],AX
```

Symbols

When on (the default), symbols are used to indicate variables; when off, addresses or offsets from register values are shown for variables. This option only affects assembly mode. The sample default display at the beginning of this section shows the appearance of the screen when the option is on. The following display shows the appearance of the same code when the option is off:

```
27:                     name = gets(namebuf);
32AF:003E 8D46DE          LEA     AX,Word Ptr [BP-22]
32AF:0041 50              PUSH    AX
32AF:0042 E89C03          CALL    03E1
32AF:0045 83C402          ADD     SP,02
32AF:0048 8946DA          MOV     Word Ptr [BP-26],AX
```

Bytes Coded

When on (the default), both the instructions and the bytes for each instruction are shown; when off, only the instructions are shown. This option only affects assembly mode. The sample default display at the beginning of this section shows the appearance of the screen when the option is on. The following display shows the appearance of the same code when the option is off:

```
27:                     name = gets(namebuf);
32AF:003E LEA             AX,Word Ptr [namebuf]
32AF:0041 PUSH            AX
32AF:0042 CALL            _gets (03E1)
32AF:0045 ADD             SP,02
32AF:0048 MOV             Word Ptr [name],AX
```

Registers

When on, the register window is displayed; when off (the default), the register window is not displayed. You can also turn on the register window by pressing F2, as indicated on the right side of the menu.

Case Sense	When on (the default), the CodeView debugger assumes that symbol names are case sensitive (each lowercase letter is different from the corresponding uppercase letter); when off, symbol names are not case sensitive. The C language is normally case sensitive, so you will probably want to leave this option on. However, you may want to turn it off when debugging assembly-language programs.

3.2.3.7 Using the Calls Menu

The Calls menu is different from other menus in that its contents and size change, depending on the status of your program. The Calls menu is shown in Figure 3.8.

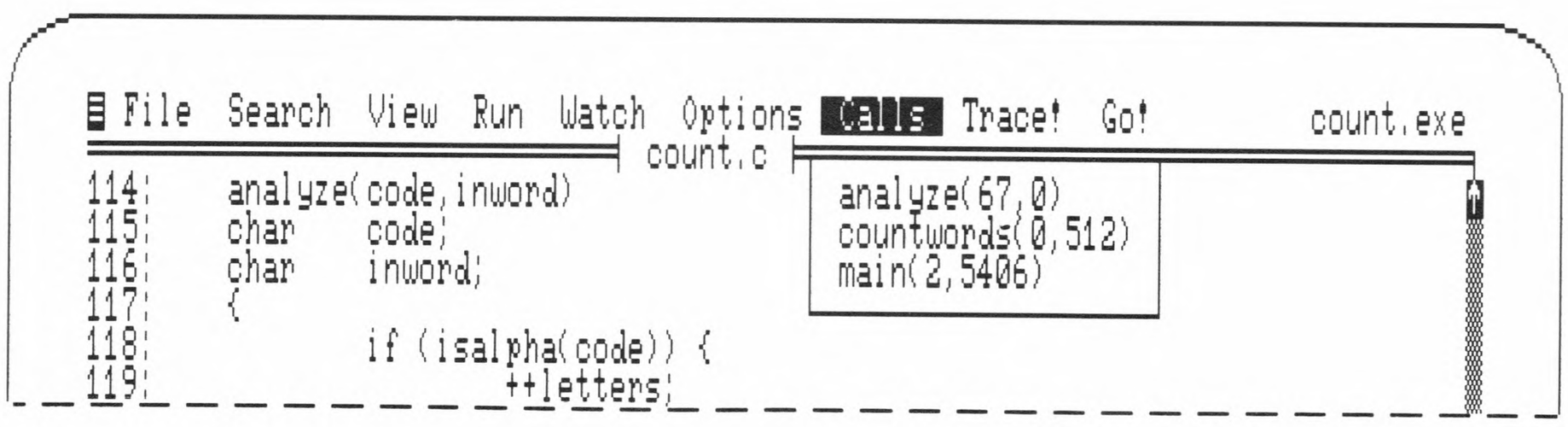


Figure 3.8 The Calls Menu

Like other menus, the Calls menu can be opened by pressing the ALT key and the first letter of the menu title (ALT-C). However, you cannot make selections from the Calls menu by pressing the ALT key with the first letter of your selection. You must use the UP ARROW or DOWN ARROW keys to move to your selection, then press the ENTER key. You can also use the mouse to open and select from the Calls menu. Usually the menu is used to view the current functions rather than to actually make selections.

The Calls menu shows the current function and the trail of functions from which it was called. The current function is always at the top. The function from which the current function was called is directly below. Other active functions are shown in the reverse order in which they were called. With C programs, the bottom function will always be **main**.

The current value of each argument, if any, is shown in parentheses following the function. The menu expands to accommodate the arguments of the widest function. Arguments are shown in the current radix (the default is decimal). If there are more active functions than will fit on the screen, or if the function arguments are too wide, the display will be truncated. The Stack Trace dialog command (**K**) shows all the functions and arguments regardless of the size of the display.

If you want to view code at the point where one of the functions was called, select the function below the one you want to view. The cursor will move to the calling source line (in source mode) or to the calling instruction (in assembly mode). In other words, the cursor will indicate the calling location in the selected function where the next-level function was called. If you select the current (top-level) function, the cursor moves to the current location in that function.

For example, if the Calls menu shown in Figure 3.8 appears and you select `countwords`, the cursor will move to the line in function **countwords** where function **analyze** is called.

3.2.4 Using the Help System

The CodeView on-line-help system uses tree-structured menus to give you quick access to help screens on a variety of subjects. The system is organized by subject so that you can reach a help screen on any topic with a miminum of keystrokes.

The help file is called **CV.HLP**. It must be present in the current directory or in one of the directories specified with the MS-DOS **PATH** command. If the help file is not found, the CodeView debugger will still operate, but you will not be able to use the help system. An error message will appear if you try to use a help command.

When you request help, either by pressing the F1 key or by selecting Help from the View menu, the top-level help menu appears. You select one of the topics listed by pressing the highlighted first letter of the menu title. You can also select a topic by pointing to it with the mouse and clicking either button.

Note

For a quick summary of dialog commands, you can use the dialog version of the Help command (**H**). This is the only help available in sequential mode. It is completely different from the on-line-help system available with window commands.

When you select a topic, the help screen on that subject may appear immediately, or in some cases, a second menu screen will appear. Keep selecting topics until you reach the screen you want.

In addition to menu titles, you can select four special commands from any help screen. The keys that select these commands are always shown at the top of the screen. You can also point and click with the mouse to select these commands. The commands are listed below:

Command	Action
PGUP	Returns to the previous help screen or menu.
PGDN	Proceeds to the next-level screen. This command is intended for help topics that consist of more than one screen of text. If you press this key on a menu screen, the top leftmost selection is chosen as the default. The CodeView debugger sounds a warning if there is no lower-level screen.
HOME	Returns to the top-level menu. The debugger sounds a warning if you are already at the top level.
END	Returns to the debugging screen.

When you use a selection letter to select a topic that has more than one screen of information, you can press the selection letter again to get the next screen. This has the same effect as pressing the PGDN key. For example, if you press W to select Watch from a menu, the first screen of watch information appears. You can press W or PGDN to get to the second watch screen.

3.3 Using Sequential Mode

Sequential mode is required if you have neither an IBM Personal Computer nor a closely compatible computer. In sequential mode, the CodeView debugger works much like its predecessors, the Microsoft Symbolic Debugging Utility (**SYMDEB**) and the MS-DOS **DEBUG** utility.

In sequential mode, the CodeView debugger's input and output always move down the screen from the current location. When the screen is full, the old output scrolls off the top of the screen to make room for new output appearing at the bottom. You can never return to examine previous commands once they scroll off, but in many cases, you can reenter the command to put the same information on the screen again.

Most window commands cannot be used in sequential mode. However, the following function keys, which are used as commands in window mode, are also available in sequential mode:

Command	Action
F1	Displays a command-syntax summary. This is equivalent to the Help (**H**) dialog command. It is different from the on-line-help system accessed by the F1 key in window mode.
F2	Displays the registers. This is equivalent to the Register (**R**) dialog command.
F3	Toggles between source and assembly modes. If the current mode is source (displaying source lines as output to the Trace, Program Step, and Go commands), the system switches to assembly mode (displaying assembly-language instructions). If the current mode is assembly, the system switches to source. This is equivalent to using the Set Assembly (**S**–) and Set Source (**S**+) dialog commands.
F4	Switches to the output screen, which shows the output of your program. Press any key to return to the CodeView debugging screen. This is equivalent to the Screen Exchange (\) dialog command.
F5	Executes from the current instruction until a breakpoint or the end of the program is encountered. This is equivalent to the Go dialog command (**G**) with no argument.

F8	Executes the next source line in source mode, or the next instruction in assembly mode. If the source line or instruction contains a function, procedure, or interrupt call, the CodeView debugger executes the first source line or instruction of the call and is ready to execute the next source line or instruction within the call. This is equivalent to the Trace (**T**) dialog command.
F9	Sets or clears a breakpoint at the current program location. If the current program location has no breakpoint, one is set. If current location has a breakpoint, it is removed. This is equivalent to the Breakpoint Set (**BP**) dialog command with no argument.
F10	Executes the next source line in source mode, or the next instruction in assembly mode. If the source line or instruction contains a function, procedure, or interrupt call, the call is executed to the end and the CodeView debugger is ready to execute the line or instruction after the call. This is equivalent to the Program Step (**P**) dialog command.

The CodeView Watch (**W**), Watchpoint (**WP**), and Tracepoint (**TP**) commands work in sequential mode, but since there is no watch window, the watch statements are not shown. You must use the Watch List command (**W**) to examine watch statements and watch values. See Chapter 8, "Managing Watch Statements," for information on Watch Statement commands.

All the CodeView commands that affect program operation (such as Trace, Go, and Breakpoint Set) are available in sequential mode. Any debugging operation that can be done in window mode can also be done in sequential mode.

Chapter 4
Using Dialog Commands

4.1 Introduction 67
4.2 Entering Commands and Arguments 67
4.2.1 Using Special Keys 67
4.2.2 Using the Command Buffer 68
4.3 Format for CodeView Commands and Arguments 69
4.4 C Expressions 70
4.4.1 Identifiers 72
4.4.2 Constant Numbers 73
4.4.3 Registers 74
4.4.4 Addresses 75
4.4.5 Address Ranges 76
4.4.6 Line Numbers 77
4.4.7 Strings 78

4.1 Introduction

CodeView dialog commands can be used in sequential mode or from the dialog window. In sequential mode, they are the primary method of entering commands. In window mode, dialog commands are used to enter commands that require arguments or that do not have corresponding window commands.

Many window commands have duplicate dialog commands. Generally, the window version of a command is more convenient, while the dialog version is more powerful. For example, to set a breakpoint on a source line in window mode, put the cursor on the source line and press F9, or point to the line and click the left mouse button. The dialog version of the command (**BP**) requires more keystrokes, but it allows you to specify an address, a pass count, and a string of commands to be taken whenever the breakpoint is encountered.

The rest of this chapter explains how to enter dialog commands and how to specify command arguments.

4.2 Entering Commands and Arguments

Dialog commands are entered at the CodeView prompt (>). Type the command and arguments, then press the ENTER key.

In window mode, you can enter commands regardless of whether or not the cursor is at the CodeView prompt. If the cursor is in the display window, the text you type will appear after the prompt in the dialog window, even though the cursor remains in the display window.

4.2.1 Using Special Keys

While entering dialog commands or viewing output from commands, you can use the following special keys:

Key	Action
CONTROL-C	Stops the current output or cancels the current command line. For example, if you are watching a long display from a Dump command, you can press CONTROL-C to interrupt the output and return to the

CodeView prompt. If you make a mistake while entering a command, you can press CONTROL-C to cancel the command without executing it. A new prompt appears and you can reenter the command.

CONTROL-S — Pauses during output of a command. You can press any key to continue output. For example, if you are watching a long display from a Dump command, you can press CONTROL-S when a part of the display that you want to examine more closely appears. Then press any key when you are ready for the output to continue scrolling.

BACKSPACE — Deletes the previous character on the command line and moves the cursor back one space. For example, if you make an error while typing a command, you can use the BACKSPACE key to delete the characters back to the error, then retype the rest of the command.

4.2.2 Using the Command Buffer

In window mode, the CodeView debugger has a command buffer where the last 4K (4096 bytes) of commands and command output are stored. This amounts to approximately three screens of text, depending on the length of your commands and output. The command buffer is not available in sequential mode.

When the cursor is in the dialog window, you can scroll up or down to view the commands you have entered earlier in the session. The commands for moving the cursor and scrolling through the buffer are explained in sections 3.2.1.1 and 3.2.2.1.

Scrolling through the buffer is particularly useful for viewing the output from commands, such as Dump or Examine Symbols, whose output may scroll off the top of the dialog window.

If you have scrolled through the dialog buffer to look at previous commands and output, you can still enter new commands. When you type a command, it will appear to be overwriting the previous line where the cursor is located, but when you press the ENTER key, the new command will be entered at the end of the buffer. For example, if you enter a command while the cursor is at the start of the buffer, and then scroll to the end of the buffer, you will see the command you just entered. If you scroll back to the point where you entered the command, you will see the original characters rather than the characters you typed over the originals.

When you start the debugger, the buffer is empty except for the copyright message. As you enter commands during the session, the buffer is gradually filled from the bottom to the top. If you have not filled the entire buffer and you press the HOME key to go to the top of the buffer, you will not see the first commands of the session. Instead you will see blank lines, since there is nothing at the top of the buffer.

4.3 Format for CodeView Commands and Arguments

The CodeView command format is similar to the format of previous Microsoft debuggers, **SYMDEB** and **DEBUG**. However, some features, particularly operators and expressions, are different. The general format for CodeView commands is shown below:

command [[*arguments*]] [[;*command2*]]

The *command* is a one-, two-, or three-character command name, and *arguments* are C expressions that represent values or addresses to be used by the command. Any combination of uppercase and lowercase letters can be used in commands. Since arguments consist of C expressions, they are normally case sensitive. Usually, the first *argument* can be placed immediately after *command* with no space separating the two.

The number of arguments required or allowed with each command varies. If a command takes two or more arguments, you must separate the arguments with spaces. A semicolon (;) can be used as a command separator if you want to specify more than one command on a line.

As a convention for clarity, examples in this manual have uppercase letters for commands and lowercase letters for arguments.

■ Examples

```
>DB 100 200       ;* Example 1

>U label1         ;* Example 2

>U label2 ;DB     ;* Example 3
```

In Example 1, DB is the first command (for the Dump Bytes command in this case). The arguments to the command are 100 and 200. The second

command on this line is the Comment command (*). A semicolon is used to separate the two commands. The Comment command is used throughout the rest of the manual to number examples.

In Example 2, `U` is the command letter (for the Unassemble command) and `label1` is a command argument. Again the Comment command labels the example.

Example 3 consists of three commands, separated by semicolons. The first is the Unassemble command (`U`) with `label2` as an argument. The second is the Dump Bytes command (`DB`) with no arguments. The third is the Comment command.

4.4 C Expressions

CodeView arguments always consist of C expressions. Expressions can include identifiers (also called symbols), constant numbers, registers, and operators. Expressions evaluate to 8-, 16-, or 32-bit values that can be used as numbers or addresses by CodeView commands.

The CodeView debugger uses a subset of operators consisting of the most commonly used C operators and one additional CodeView operator, the colon (:). The CodeView operators are listed in Table 4.1 in order of precedence.

Table 4.1

CodeView Operators

Precedence	Operators
(Highest)	
1	() [] -> .
2	! ~ -[a] (*type*) ++ -- *[b] &[c] **sizeof**
3	*[b] / % :
4	+ -[a]
5	< > <= >=
6	== !=

Table 4.1 *(continued)*

Precedence	Operators
7	&&
8	||
9	= += -= *= /= %=
(Lowest)	

[a] The minus sign with precedence 2 is the unary minus indicating the sign of a number, while the minus sign with precedence 4 is a binary minus indicating subtraction.

[b] The asterisk with precedence 2 is the pointer operator, while the asterisk with precedence 3 is the multiplication operator.

[c] The ampersand with precedence 2 is the address-of operator. The ampersand as a bitwise-AND operator is not supported by the CodeView debugger.

See the *Microsoft C Compiler Language Reference* for a description of how C operators can be combined with identifiers and constants to form expressions.

The colon operator (:) is the only CodeView operator that does not come from C. It acts as a *segment*:*offset* separator, as described in Section 4.4.3, "Registers."

In the CodeView debugger, the period (.) has its normal use as a member selection operator, but it also has an extended use as a specifier of local variables in parent functions. The syntax is shown below:

function.*variable*

The *function* must be a higher-level function and the *variable* must be a local variable within the specified function. The *variable* cannot be a register variable. For example, you can use the expression `main.argc` to refer to the local variable `argc` when you are in a function that has been called by `main`.

The *type* operator (used in type casting) can be any of the predefined C types. The CodeView debugger limits pointer types to one level of indirection. For example, `(char *)sym` is accepted, but `(char **)sym` is not.

When a C expression is used as an argument with a command that takes multiple arguments, the expression should not have any internal spaces. For example, `count+6` is allowed, but `count + 6` may be interpreted as three separate arguments. Some commands (such as the Display Expression command) do permit spaces in expressions.

Note

When you try to use a variable in an expression in a case where that variable is not defined, the CodeView debugger displays the message `UNKNOWN SYMBOL`. For example, the message appears if you try to reference a local variable outside the function where the variable is defined.

Sections 4.4.1–4.4.5 tell how to specify arguments using CodeView expressions.

4.4.1 Identifiers

■ Syntax

name

An identifier is a name that represents a register, an absolute value, a segment address, or an offset address. Identifiers (also called symbols) follow the naming rules of the C compiler. Note that while CodeView command letters are not case sensitive, symbols given as arguments are case sensitive (unless you have turned off case sensitivity with the Case Sense selection from the Options menu).

In assembly-language output or in output from the Examine Symbols command, the CodeView debugger displays some symbol names in the object-code format produced by the Microsoft C Compiler. This format includes a leading underscore. For example, the function `main` will be displayed as `_main`. Only global labels (such as procedure names) are shown in this format. You do not need to include the underscore when specifying such a symbol in CodeView commands. Labels within library routines are sometimes displayed with a double underscore (`__chkstk`). You must use leading underscores when accessing these labels with CodeView commands.

4.4.2 Constant Numbers

■ Syntax

digits	Decimal format
0*digits*	Octal format
0x*digits*	Hexadecimal format
0n*digits*	Alternate decimal format

Numbers used in CodeView commands represent integer constants. They are made up of octal, decimal, or hexadecimal digits, and are entered in the current input radix. The C-language format for entering numbers of different radixes can be used to override the input radix.

The default radix for the C-language version of the CodeView debugger is decimal. However, you can use the Radix command (**N**) to specify an octal or hexadecimal radix, as explained in Section 11.4, "Radix Command."

If the current radix is 16 (hexadecimal) or 8 (octal), you can enter decimal numbers in the special CodeView format **0n***digits*. For example, enter 21 decimal as `0n21`.

With radix 16, it is possible to enter a value or argument that could be interpreted either as an identifier or as a hexadecimal number. The CodeView debugger resolves the ambiguity by searching first for a symbol (identifier) with that name. If no symbol is found, the debugger interprets the value as a hexadecimal number. If you want to enter a number that overrides an existing symbol, use the hexadecimal format (0x*digits*).

For example, if you enter `abc` as an argument when the program contains a variable or function named `abc`, the CodeView debugger interprets the argument as the symbol. If you want to enter `abc` as a number, enter it as `0xabc`.

Table 4.2 shows how a sample number (63 decimal) would be represented in each radix.

Table 4.2

CodeView Radix Examples

Input Radix	Octal	Decimal	Hexadecimal
8	77	On63	Ox3F
10	077	63	Ox3F
16	077	On63	3F

4.4.3 Registers

■ Syntax

[[@]]*register*

You can specify a register name if you want to use the current value stored in the register. Registers are rarely needed in C source debugging, but they are used frequently for assembly-language debugging.

When you specify an identifier, the CodeView debugger first checks the symbol table for a symbol with that name. If the debugger does not find a symbol, it checks to see if the identifier is a valid register name. If you want the identifier to be considered a register, regardless of any name in the symbol table, use the at sign (@) as a prefix before the register name. For example, if your program has a symbol called AX, you could specify @AX to refer to the **AX** register. You can avoid this problem entirely by making sure that identifier names in your program do not conflict with register names.

The register names known to the CodeView debugger are shown in Table 4.3.

Table 4.3

Registers

Type	Names			
16-bit general purpose	**AX**	**BX**	**CX**	**DX**
8-bit high registers	**AH**	**BH**	**CH**	**DH**
8-bit low registers	**AL**	**BL**	**CL**	**DL**

Table 4.3 *(continued)*

Type	Names			
16-bit segment	**CS**	**DS**	**SS**	**ES**
16-bit pointer	**SP**	**BP**	**IP**	
16-bit index	**SI**	**DI**		

4.4.4 Addresses

■ Syntax

[[*segment*:]]*offset*

Addresses can be specified in the CodeView debugger through use of the colon operator as a *segment*:*offset* connector. Both the *segment* and the *offset* are made up of expressions.

A full address has a *segment* and an *offset*, separated by a colon. A partial address has just an *offset*; a default segment is assumed. The default *segment* varies, depending on the command with which the address is used. Commands that refer to data (Dump, Enter, Watch, and Tracepoint) use the contents of the **DS** register. Commands that refer to code (Assemble, Breakpoint Set, Go, Unassemble, and View) use the contents of the **CS** register.

Full addresses are seldom necessary in C debugging. Occasionally they may be convenient for referring to addresses outside the program, such as BIOS (basic input/output system) or MS-DOS addresses. A full address is equivalent to a C expression cast as a far pointer to a **char**. For example, the full address `1234:5678` means `(char far *)1234:5678`.

■ Examples

```
>DB 100                    ;* Example 1

>DB array[count]           ;* Example 2

>DB label+10               ;* Example 3

>DB 0xB800:0xFF            ;* Example 4
```

In Example 1, the Dump Bytes command (**DB**) is used to dump memory starting at offset address `100`. Since no segment is given, the data segment (the default for Dump commands) is assumed.

In Example 2, the Dump Bytes command is used to dump memory starting at the address of the variable `array[count]`.

In Example 3, the Dump Bytes command is used to dump memory starting at a point 10 bytes beyond the symbol `label`.

In Example 4, the Dump Bytes command is used to dump memory at the absolute address having the segment value `0xB800` and the offset address `0xFF`.

4.4.5 Address Ranges

■ Syntax

startaddress endaddress
startaddress **L** *count*

A range is a pair of memory addresses that bound a sequence of contiguous memory locations.

You can specify a range in two ways. One way is to give the start and end points. In this case the range covers *startaddress* to *endaddress*, inclusive. If a command takes a range, but you do not supply a second address, the CodeView debugger usually assumes the default range. Each command has its own default range (the most common default range is 128 bytes).

You can also specify a range by giving its starting point and the number of objects you want included in the range. This type of range is called an object range. In specifying an object range, *startaddress* is the address of the first object in the list, **L** indicates that this is an object range rather than an ordinary range, and *count* specifies the number of objects in the range.

The size of the objects is the size taken by the command. For example, the Dump Bytes command (**DB**) has byte objects, the Dump Words command (**DW**) has words, the Unassemble (**U**) command has instructions, and so on.

■ Examples

```
>DB buffer                ;* Example 1

>DB buffer buffer+20      ;* Example 2

>DB buffer L 20           ;* Example 3

>U label-30 label         ;* Example 4
```

Example 1 dumps a range of memory starting at `buffer`. Since the end of the range is not given, the default size (128 bytes for the Dump Bytes command) is assumed.

Example 2 dumps a range of memory starting at `buffer` and ending at `buffer+20` (the point 20 bytes beyond `buffer`).

Example 3 uses an object range to dump the same range as in Example 2. The `L` indicates that the range is an object range, and `20` is the number of objects in the range. Each object has a size of one byte, since that is the command size.

Example 4 uses the Unassemble command (`U`) to list the assembly-language statements starting 30 instructions before `label` and continuing to `label`.

4.4.6 Line Numbers

■ Syntax

.[[*filename*:]]*linenumber*

The address corresponding to a source line number can be specified as *linenumber* prefixed with a period (.). The CodeView debugger assumes the source line is in the current source file unless you specify the optional *filename* followed by a colon and the line number.

The CodeView debugger displays an error message if *filename* does not exist, or if no source line exists for the specified number.

■ Examples

```
>V .100                ;* Example 1

>V .sample.c:10        ;* Example 2
```

Example 1 uses the View command (**V**) to display code starting at the address that corresponds to source line `100`. Since no file is indicated, the current source file is assumed.

Example 2 uses the View command to display source code starting at the address that corresponds to source line `10` of the source file `sample.c`.

4.4.7 Strings

■ Syntax

"*null-terminated-string*"

Strings can be specified as expressions in the C format. You can use C escape characters within strings. For example, double quotation marks within a string must be specified with the escape character \".

■ Example

```
>EA message "This \"string\" is okay."
```

The example uses the Enter ASCII command (**EA**) to enter the given string into memory starting at the address of the variable `message`.

Chapter 5
Executing Code

5.1 Introduction 81
5.2 Trace Command 82
5.3 Program Step Command 84
5.4 Go Command 87
5.5 Execute Command 90
5.6 Restart Command 91

5.1 Introduction

The Trace (**T**), Program Step (**P**), Go (**G**), and Execute (**E**) commands are used to execute code within a program. Among the differences between them is the size of step executed by each command. The commands and their step sizes are listed below:

Command	Action
Trace (**T**)	Executes the current source line in source mode, or the current instruction in assembly mode; traces into functions, procedures, or interrupts
Program Step (**P**)	Executes the current source line in source mode, or the current instruction in assembly mode; steps over functions, procedures, or interrupts
Go (**G**)	Executes the current program
Execute (**E**)	Executes the current program in slow motion
Restart (**L**)	Restarts the current program

In window mode, the screen is updated to reflect changes that occur when you execute a Trace, Program Step, or Go command. The highlight marking the current location is moved to the new current instruction in the display window. Values are changed, if appropriate, in the register and watch windows.

In sequential mode, the current source line or instruction is displayed after each Trace, Program Step, or Go command. The format of the display depends on the display mode. The three display modes available in sequential mode (source, assembly, and mixed) are discussed in Chapter 9, "Examining Code."

If the display mode is source (**S+**) in sequential mode, the current source line is shown. If the display mode is assembly (**S–**), the status of the registers and flags and the new current instruction are shown in the format of the Register command (see Chapter 6, "Examining Data and Expressions"). If the display mode is mixed (**S&**), the registers, the new source line, and the new instruction are all shown.

The commands that execute code are explained in sections 5.2–5.5.

Note

If you are executing a section of code with the Go or Program Step command, you can usually interrupt program execution by pressing CONTROL-BREAK or CONTROL-C. This can terminate endless loops, or it can interrupt loops that are delayed by the Watchpoint or Tracepoint commands (see Chapter 8, "Managing Watch Statements"). CONTROL-BREAK or CONTROL-C may not work if your program has a special use for either of these key combinations. If you have an IBM Personal Computer AT (or a compatible computer), you can use the SYSTEM-REQUEST key to interrupt execution regardless of your program's use of CONTROL-BREAK and CONTROL-C.

5.2 Trace Command

The Trace command executes the current source line in source mode, or the current instruction in assembly mode. The current source line or instruction is the one pointed to by the **CS** and **IP** registers. In window mode, the current instruction is shown in reverse video or in a contrasting color.

In source mode, if the current source line contains a function call, the CodeView debugger executes the first source line of the function. In assembly mode, if the current instruction contains a procedure or interrupt, the debugger executes the first instruction of the procedure or interrupt.

Use the Trace command if you want to trace into functions, procedures, or interrupts. If you want to execute calls without tracing into them, you should use the Program Step command (**P**) instead. Both commands execute MS-DOS function calls (interrupt 0x21) without tracing into them. There is no direct way to trace into MS-DOS function calls.

In source mode, the CodeView debugger will only trace into functions that have source code. For example, if the current line contains a call to the C library function **printf**, the debugger will execute the function if you are in source mode, since the source code for library functions is not available. If you are in assembly or mixed mode, the debugger will trace into the function.

Note

The Trace command (**T**) uses the hardware trace mode of the 8086 family of processors. Consequently, you can also trace instructions stored in ROM (read-only memory). However, since breakpoints cannot be set in ROM, the Program Step command (**P**) will not work. Using it in ROM has the same effect as using the Go command (**G**).

■ Mouse

To execute the Trace command with the mouse, point to Trace on the menu bar and click the left button.

■ Keyboard

To execute the Trace command with a keyboard command, press the F8 key. This works in both window and sequential modes.

■ Dialog

To execute the Trace command using a dialog command, enter a command line with the following syntax:

T [[*count*]]

If the optional *count* is specified, the command executes *count* times before stopping.

■ Examples

The following examples all show the Trace command in sequential mode. In window mode, there would be no output from the commands, but the display would be updated to show changes caused by the command.

```
>S+          ;* Example 1
source
>.
73:                          analyze(code,inword);
```

```
>T 4
90:    char  code;
92:          {
94:              ++letters;
95:              if (strchr("AEIOUaeiou",code) || (strchr("yY",code) && !inword))
>
```

Example 1 sets the display mode to source, then uses the Source Line command to display the current source line. (See Chapter 9, "Examining Code," for a further explanation of the Set Source and Source Line commands.) Note that the current source line calls the function `analyze`. The Trace command is then used to execute the next four source lines. These lines will be the first four lines of the function `analyze`.

```
>S-         ;* Example 2
assembly
>T
AX=0001  BX=0001  CX=0000  DX=0000  SP=1900  BP=1908  SI=04BA  DI=1946
DS=3BB1  ES=3BB1  SS=3BB1  CS=36C0  IP=0237  NV UP EI PL NZ NA PO NC
36C0:0237 50              PUSH      AX
>
```

Example 2 sets the display mode to assembly and traces the current instruction. This example and the next example are the same as the examples of the Program Step command in Section 5.3. The Trace and Program Step commands are only different if, when the command is executed, the current instruction is a function, procedure, or interrupt call.

```
>S&         ;* Example 3
mixed
>T
AX=0043  BX=0043  CX=025C  DX=0000  SP=1900  BP=1904  SI=04BA  DI=1952
DS=5BE4  ES=5BE4  SS=5BE4  CS=56F3  IP=026E  NV UP EI PL NZ NA PO NC
92:                ++letters;
56F3:026E FF067201      INC      Word Ptr [_letters (0172)]     DS:0172=0000
>
```

Example 3 sets the display mode to mixed and traces the current instruction.

5.3 Program Step Command

The Program Step command executes the current source line in source mode, or the current instruction in assembly mode. The current source line or instruction is the one pointed to by the **CS** and **IP** registers. In window mode, the current instruction is shown in reverse video or in a contrasting color.

In source mode, if the current source line contains a function call, the CodeView debugger executes the entire function and is ready to execute the line after the function call. In assembly mode, if the current instruction contains a procedure or interrupt, the debugger executes the entire procedure or interrupt and is ready to execute the next instruction after the procedure or interrupt call.

Use the Program Step command if you want to execute over function, procedure, and interrupt calls. If you want to trace into calls, you should use the Trace command (**T**) instead. Both commands execute MS-DOS function calls (interrupt 0x21) without tracing into them. There is no direct way to trace into MS-DOS function calls.

■ Mouse

To execute the Program Step command with the mouse, point to Trace on the menu bar and click the right button.

■ Keyboard

To execute the Program Step command with a keyboard command, press the F10 key. This works in both window and sequential modes.

■ Dialog

To execute the Program Step command using a dialog command, enter a command line with the following syntax:

P [[*count*]]

If the optional *count* is specified, the command executes *count* times before stopping.

■ Examples

The examples show the Program Step command in sequential mode. In window mode, there would be no output from the commands, but the display would be updated to show changes caused by the command.

```
>S+       ;* Example 1
source
>.
73:                              analyze(code,inword);
>P 4
74:                              inword = TRUE;
75:                              ++words;
76:                              ++characters;
78:                         }
```

Example 1 sets the display mode to source, then uses the Source Line command to display the current source line. (See Chapter 9, "Examining Code," for a further explanation of the Set Source and Source Line commands.) Note that the current source line calls the function `analyze`. The Program Step command is then used to execute the next four source lines. The first program step executes the entire function `analyze`, and the next three steps execute the lines immediately after the function.

```
>S-       ;* Example 2
assembly
>P
AX=0001  BX=0001  CX=0000  DX=0000  SP=1900  BP=1908  SI=04BA  DI=1946
DS=3BB1  ES=3BB1  SS=3BB1  CS=36C0  IP=0237  NV UP EI PL NZ NA PO NC
36C0:0237 50              PUSH      AX
>
```

Example 2 sets the display mode to assembly and steps through the current instruction. This example and the next example are the same as the examples of the Trace command in Section 5.2. The Trace and Program Step commands are only different if, when the command is executed, the current instruction is a function, procedure, or interrupt call.

```
>S&       ;* Example 3
mixed
>P
AX=0043  BX=0043  CX=025C  DX=0000  SP=1900  BP=1904  SI=04BA  DI=1952
DS=5BE4  ES=5BE4  SS=5BE4  CS=56F3  IP=026E  NV UP EI PL NZ NA PO NC
92:                  ++letters;
56F3:026E FF067201       INC       Word Ptr [_letters (0172)]      DS:0172=0000
>
```

Example 3 sets the display mode to mixed and steps through the current instruction.

5.4 Go Command

The Go command starts execution at the current address. There are two variations of the Go command. One simply starts execution and continues to the end of the program or until a breakpoint is encountered. The other variation is a Goto command, in which a destination is given with the command.

The Go command will stop execution when a breakpoint set earlier with the Breakpoint Set (**BP**), Watchpoint (**BP**), or Tracepoint (**TP**) command is encountered. If the command is given in the Goto form, the execution will stop before the destination is reached if a previously set breakpoint is encountered first.

If a destination address is given, but never encountered (for example, if the destination is on a program branch that is never taken), the CodeView debugger executes to the end of the program.

If you enter the Go command and the debugger does not encounter a breakpoint, the entire program is executed and the following message is displayed:

```
Program terminated normally (number)
```

The *number* in parentheses is the value returned by the program (sometimes called the exit or "errorlevel" code).

■ Mouse

To execute the Go command with no destination, point to Go on the menu bar and press either button.

To execute the Goto variation of the Go command, point to the source line or instruction to which you wish to go; then press the right button. The highlight marking the current location will move to the source line or instruction you pointed to (unless a breakpoint is encountered first). The CodeView debugger will sound a warning and take no action if you try to go to a comment line or other source line that does not correspond to code.

If the line you wish to go to is in another module, you can use the Load command from the Files menu to load the source file for the other module. Then point to the destination line and press the right button.

■ Keyboard

To execute the Go command with no destination using a keyboard command, press the F5 key. This works in both window and sequential modes.

To execute the Goto variation of the Go command, move the cursor to the source line or instruction you wish to go to. If the cursor is in the dialog window, first press the F6 key to move the cursor to the display window. When the cursor is at the appropriate line in the display window, press the F7 key. The highlight marking the current location will move to the source line or instruction you pointed to (unless a breakpoint is encountered first). The CodeView debugger will sound a warning and take no action if you try to go to a comment line or other source line that does not correspond to code.

If the line you wish to go to is in another module, you can use the Load command from the Files menu to load the source file for the other module. Then move the cursor to the destination line and press the F7 key.

■ Dialog

To execute the Go command using a dialog command, enter a command line with the following syntax:

G [[*breakaddress*]]

If the command is given with no argument, execution continues until a breakpoint or the end of the program is encountered.

The Goto form of the command can be given by specifying *breakaddress*. The *breakaddress* can be given as a symbol, a line number, or an address in the *segment*:*offset* format. If the offset address is given without a segment, the address in the **CS** register is used as the default segment. If you give *breakaddress* as a line number, but the corresponding source line is a comment, declaration, or blank line, the following message appears:

```
No code at this line number
```

■ Examples

The following examples show the Go command in sequential mode. In window mode, there would be no output from the commands, but the display would be updated to show changes caused by the command.

```
>G                    ;* Example 1

Program terminated normally (0)
>
```

Example 1 passes control to the instruction pointed to by the current values of the **CS** and **IP** registers. No breakpoint is encountered, so the CodeView debugger executes to the end of the program, where it prints a termination message and the exit code returned by the program (0 in the example).

```
>S+                ;* Example 2
source
>G analyze
22:     int   argc;
>
```

In Example 2, the display mode is first set to source (**S+**). See Chapter 9, "Examining Code," for information on setting the display mode. When the Go command is entered, the CodeView debugger starts program execution at the current address and continues until it reaches the start of the function `analyze`.

```
>S&          ;* Example 3
mixed
>G .38
AX=13F3  BX=13EA  CX=0019  DX=0000  SP=130E  BP=133A  SI=04BA  DI=1344
DS=5DA8  ES=5DA8  SS=5DA8  CS=58B4  IP=004B  NV UP EI PL NZ NA PO NC
38:              if ((stream = fopen(name,"rb")) == NULL) return (1);
58B4:004B B86E00          MOV       AX,006E
>
```

Example 3 passes execution control to the program at the current address and executes to the address of source line 38. If the address with the breakpoint is never encountered (for example, if the program has less than 38 lines, or if the breakpoint is on a program branch that is never taken), the CodeView debugger executes to the end of the program.

```
>S-          ;* Example 4
assembly
>G 0x2A8
AX=0049  BX=0049  CX=028F  DX=0000  SP=12F2  BP=12F6  SI=04BA  DI=1344
DS=5DAF  ES=5DAF  SS=5DAF  CS=58BB  IP=02A8  NV UP EI PL NZ NA PE NC
58BB:02A8 98              CBW
>
```

Example 4 executes to address CS:0x2A8. Since no segment address is given, the **CS** register is assumed.

5.5 Execute Command

The Execute command is similar to the Go command with no arguments, except that it executes in slow motion (several source lines per second). Execution starts at the current address and continues to the end of the program or until a breakpoint, tracepoint, or watchpoint is reached. You can also stop automatic program execution by pressing any key or a mouse button.

■ Mouse

To execute code in slow motion with the mouse, point to Run on the menu bar, press a mouse button and drag the highlight down to the Execute selection, then release the button.

■ Keyboard

To execute code in slow motion with a keyboard command, press ALT-R to open the Run menu, then press ALT-E to select Execute.

■ Dialog

To execute code in slow motion using a dialog command, enter a command line with the following syntax:

E

You cannot set a destination for the Execute command as you can for the Go command.

In sequential mode, the output from the Execute command depends on the display mode (source, assembly, or mixed). In assembly or mixed mode, the command executes one instruction at a time. The command displays the current status of the registers and the instruction. In mixed mode, it will also show a source line if there is one at the instruction. In source mode, the command executes one source line at a time, displaying the lines as it executes them.

Important

The Execute command has the same command letter (**E**) as the Enter command. If the command has at least one argument, it is interpreted as Enter; if not, it is interpreted as Execute.

5.6 Restart Command

The Restart command restarts the current program. The program is ready to be executed just as if you had restarted the CodeView debugger. Any existing breakpoints or watch statements are retained. The pass count for all breakpoints is reset to 1. Any program arguments are also retained, though they can be changed with the dialog version of the command.

The Restart command can only be used to restart the current program. If you wish to load a new program, you must exit and restart the CodeView debugger with the new program name.

■ Mouse

To restart the program with the mouse, point to Run on the menu bar, press a mouse button and drag the highlight down to the Restart or the Start selection, then release the button. The program will be restarted. If the Restart selection is chosen, the program will be ready to start executing from the beginning. If the Start selection is chosen, the program starts executing from the beginning and continues until a breakpoint or the end of the program is encountered.

■ Keyboard

To restart the program with a keyboard command, press ALT-R to open the Run menu, then press either ALT-R to select Restart or ALT-S to select Start. The program will be restarted. If the Restart selection is chosen, the program will be ready to start executing from the beginning. If the Start selection is chosen, the program starts executing from the beginning and continues until a breakpoint or the end of the program is encountered.

■ Dialog

To restart the program with a dialog command, enter a command line with the following syntax:

L [[*arguments*]]

When you restart using the dialog version of the command, the program will be ready to start executing from the beginning. If you want to restart with new program arguments, you can give optional *arguments*. You cannot specify new arguments with the mouse or keyboard version of the command.

Note

The command letter "L" is a mnemonic for Load, but the command should not be confused with the Load selection from the File menu, since that selection only loads a source file without restarting the program.

■ Examples

```
>L        ;* Example 1
>

>L 6      ;* Example 2
>
```

Example 1 restarts the current executable file, retaining any breakpoints, watchpoints, tracepoints, and arguments. Example 2 restarts the current executable file, but with 6 as the new program argument.

Chapter 6
Examining Data and Expressions

6.1 Introduction 95
6.2 Display Expression Command 95
6.3 Examine Symbols Command 100
6.4 Dump Commands 103
6.4.1 Dump 105
6.4.2 Dump Bytes 106
6.4.3 Dump ASCII 106
6.4.4 Dump Integers 107
6.4.5 Dump Unsigned Integers 108
6.4.6 Dump Words 109
6.4.7 Dump Double Words 109
6.4.8 Dump Short Reals 110
6.4.9 Dump Long Reals 111
6.4.10 Dump 10-Byte Reals 112
6.5 Register Command 113
6.6 8087 Command 115

6.1 Introduction

The CodeView debugger provides several commands for examining different kinds of data, including expressions, variables, memory, and registers. The data-evaluation commands discussed in this chapter are summarized below:

Command	Action
Display Expression (**?**)	Evaluates and displays the value of symbols or expressions
Examine Symbol (**X?**)	Displays the addresses of symbols
Dump (**D**)	Displays sections of memory containing data; there are several variations for examining different kinds of data
Register (**R**)	Shows the current values of each register and each flag
8087 (**7**)	Shows the current values in the 8087 or 80287 register

6.2 Display Expression Command

The Display Expression command displays the value of CodeView expressions.

A CodeView expression can be any valid C expression consisting of numbers, symbols, addresses, type casts, or operators. The CodeView debugger supports a subset of the C operators, as explained in Chapter 4, "Using Dialog Commands." The simplest form of expression is an identifier representing a variable or label.

In addition to its primary purpose of displaying values, the Display Expression command can also set values as a side effect. For example, if you give the expression `++i` with the Display Expression command, the value of `i` will be incremented and displayed.

You can specify the format in which the values of expressions are displayed. Type a comma after the expression, followed by a **printf** type specifier. The type specifiers used in the CodeView debugger are a subset of those used by the C **printf** function. They are listed in Table 6.1.

Table 6.1
Types for printf

Character	Output Format	Sample Expression	Sample Output
d	Signed decimal integer	`?40000,d`	`-25536`
i	Signed decimal integer	`?40000,i`	`-25536`
u	Unsigned decimal integer	`?40000,u`	`40000`
o	Unsigned octal integer	`?40000,o`	`116100`
x | **X**[1]	Hexadecimal integer	`?40000,x`	`9c40`
f	Signed value in floating-point decimal format with six decimal places	`?(float)3/2,f`	`1.500000`
e | **E**[2]	Signed value in scientific-notation format with up to six decimal places (trailing zeros and decimal point are truncated)	`?(float)3/2,f`	`1.500000e+000`
g | **G**[2]	Signed value with floating-point decimal format (**f** or **F**) or scientific-notation format (**g** or **G**), whichever is more compact	`?(float)3/2,f`	`1.5`
c	Single character	`?65,c`	`A`
s	Characters printed up to the first null character	`?"String"`	`String`

[1] Hexadecimal letters are uppercase if the type is **X** and lowercase if the type is **x**.

[2] The "E" in scientific-notation numbers is uppercase if the type is **E** or **G**, lowercase if the type is **e** or **g**.

If no type specifier is given, real numbers of type **float** or **double** are displayed as if the type specifier had been given as **g**. If the numbers are signed, values of all other types are displayed as if the type specifier had been given as **d**. If the numbers are unsigned, values of all other types are displayed as if the type specifier had been given as **U**. Pointers are displayed as if the type specifier had been given as **u**.

The prefix **h** can be used with the integer type specifiers (**d**, **o**, **u**, **x**, and **X**) to specify a **short int**. The prefix **l** can be used with the same types to specify a **long int**. For example, the command `?100000,ld` produces the output `100000`. However, the command `?100000,hd` evaluates only the **short int** part of the value, producing the output `-31072`.

Note

The **n** and **p** type specifiers and the **F** and **H** prefixes are not supported by the CodeView debugger even though they are supported by the C **printf** function. See the **printf** function in the *Microsoft C Compiler Run-Time Library Reference* for more information.

■ Mouse

To display the value of an expression with the mouse, point to View on the menu bar, press a mouse button and drag the highlight down to the Evaluate selection, then release the button. A dialog box appears, asking for the expression you want to evaluate. Type the expression and press the ENTER key (or a mouse button). You can add a comma and a type specifier if you want to specify the output format. The value of the expression will appear in the dialog window.

■ Keyboard

To display the value of an expression using a keyboard command, press ALT-V to open the View menu, then press ALT-E to select Evaluate. A dialog box appears, asking for the expression you want to evaluate. Type the expression and press the ENTER key. You can add a comma and a type specifier if you want to specify the output format. The value of the expression will appear in the dialog window.

■ Dialog

To display the value of an expression using a dialog command, enter a command line with the following syntax:

? *expression*,[[*format*]]

The *expression* is any valid CodeView expression, and the optional *format* is a **printf** type specifier.

■ Examples

The examples assume that the source file contains the following variable declarations:

```
int     amount
char    *text
int     miles
char    hours
struct  {
        char name[20];
        int id;
        long class;
} student, *pstudent;

int     square (int);
```

Assume also that the program has been executed to the point where all these variables have been assigned values.

```
>? amount          ;* Example 1
500
>? amount,x
1f4
>? amount,o
764
>? &amount,X
1EE2
>
```

Example 1 displays the value of the variable `amount`, first in the default decimal format, then in hexadecimal and then in octal. Finally, the address-of operator is used to display the address where the value of `amount` is stored. Only the offset portion of the address is shown; the data segment is assumed.

```
>? 92,X            ;* Example 2
5C
>? 109*37,o
7701
>? 'T'
84
>? 118,c
v
>
```

Example 2 illustrates how the CodeView debugger can be used as a calculator. You can convert between radixes, calculate the value of constant expressions, or check ASCII equivalents.

```
>? text,X          ;* Example 3
13F3
>DA 0x13F3
3D83:13F0  Here is a string.
>? text,s
Here is a string.
>
```

Example 3 shows how to examine strings. One method is to evaluate the variable that points to the string, then dump the values at that address (the Dump command is explained in Section 6.4). A more direct method is to use the `s` type specifier.

```
>? miles           ;* Example 4
837
>? hours
14
>? miles/hours
59
>? (float)miles/hours,f
59.785714
>? (float)miles/hours,e
5.978571e+001
>
```

Example 4 displays the value of the symbols `miles` and `hours`. The two variables are then combined to calculate miles per hour. The value is calculated using integer division, then it is type cast so that real-number division can be performed. The real number is shown both in floating-point format using the `f` type specifier, and in scientific notation using the `e` type specifier.

```
>? student.id    ;* Example 5
19643
>? pstudent->id
19643
>
```

Example 5 illustrates how to display the values of members of a structure (or union).

```
>? amount          ;* Example 6
500
>? ++amount
501
>? amount=600
600
>
```

Example 6 shows how the Display Expression command can be used to change the values of variables.

```
>? square(9)       ;* Example 7
81
>
```

Example 7 shows how functions can be evaluated in expressions. The CodeView debugger executes the function `square` with an argument of `9` and displays the value returned by the function. You can only display the values of functions after you have executed into the `main` function.

6.3 Examine Symbols Command

The Examine Symbols command displays the names and addresses of symbols, and the names of modules, defined within a program. You can specify the symbol or group of symbols you want to examine by module, procedure, or symbol name.

■ Mouse

This command cannot be executed with the mouse.

■ Keyboard

This command cannot be executed with a keyboard command.

■ Dialog

To view the addresses of symbols using a dialog command, enter a command line in one of the following formats:

X*
X? [[*module*!]] [[*function.*]] [[*symbol*]] [[*]]

The syntax combinations are listed in more detail below:

Syntax	Display
X?*module*!*function.symbol*	The specified *symbol* in the specified *function* in the specified *module*
X?*module*!*function.**	All symbols in the specified *function* in the specified *module*
X?*module*!*symbol*	The specified *symbol* in the specified *module*; symbols within functions (automatic variables and statics within functions) are not found
X?*module*!*	All symbols in the specified *module*
X?*function.symbol*	The specified *symbol* in the specified *function*; looks for *function* first in the current module, then in other modules from first to last
X?*function.**	All symbols in the specified *function*; looks for *function* first in the current module, then in other modules from first to last
X?*symbol*	Looks for the specified *symbol* in this order: 1. In the current function 2. In the current module 3. In other modules, from first to last
X?*	All symbols in the current function
X*	All module names

■ Examples

In the following examples, assume that the program being examined is called `pi.exe`, and that it consists of two modules: `pi.c` and `math.c`. The `pi.c` module is a skeleton consisting only of the `main` function, while the `math.c` module has several functions. Assume that the current function is **div** within the *math* module.

```
>X*                     ;*Example 1
PI.OBJ
MATH.OBJ
C:B(chkstk)
C:B(crt0)
.
.
.
C:B(itoa)
C:B(unlink)
>
```

Example 1 lists the two modules called by the program. The library file and each of the modules called by the program are also listed.

```
>X?*                ;*Example 2
                DI          int             b
                [BP-0006]   int             quotient
                SI          int             i
                [BP-0002]   int             remainder
                [BP+0004]   int             divisor
>
```

Example 2 lists the symbols in the current function (**div**). Local variables are shown as being stored either in a register (`b` in register `DI`), or at a memory location specified as an offset from a register (`divisor` at location `[BP+0004]`).

```
>X?pi!*                 ;* Example 3
3D37:19B2 int               _scratch0       3D37:0A10 char          _p[]
3D37:2954 int               _scratch1       3D37:19B4 char          _t[]
3D37:2956 int               _scratch2       3D37:19B0 int           _q
3A79:0010 int               _main()          3A79:0010 int            main()
3D37:19B2 int                scratch0
3D37:0A10 char               p[]
3D37:2954 int                scratch1
3D37:19B4 char               t[]
3D37:2956 int                scratch2
3D37:19B0 int                q
>
```

Example 3 shows all the symbols in the the `pi.c` module.

```
>X?math!div.*    ;*Example 4
3A79:0264 int                  div()
            DI          int                 b
            [BP-0006]   int                 quotient
            SI          int                 i
            [BP-0002]   int                 remainder
            [BP+0004]   int                 divisor
>
```

Example 4 shows the symbols in the `div` function in module `math.c`. You wouldn't need to specify the module if `math.c` were the current module, but you would if the current module were `pi.c`.

Variables that are local to a function are indented under that function.

```
>X?math!arctan.s ;* Example 5
3A79:00FA int                  arctan()
            [BP+0004] int                   s
>
```

Example 5 shows one specific variable (`s`) within the `arctan` function.

6.4 Dump Commands

The CodeView debugger has several commands for dumping data from memory to the screen (or other output device). The dump commands are listed below:

Command	Command Name
D	Dump (size is the default type)
DB	Dump Bytes
DA	Dump ASCII
DI	Dump Integers
DU	Dump Unsigned Integers
DW	Dump Words

DD	Dump Double Words
DS	Dump Short Reals
DL	Dump Long Reals
DT	Dump 10-Byte Reals

■ Mouse

The Dump commands cannot be executed with the mouse.

■ Keyboard

The Dump commands cannot be executed with keyboard commands.

■ Dialog

To execute any Dump command using a dialog command, enter a command line with the following syntax:

D[[*type*]] [[*address* | *range*]]

The *type* is a one-letter specifier that indicates the type of the data to be dumped. The dump commands expect either a starting *address* or a *range* of memory. If the starting *address* is given, the commands assume a default range (usually 128 bytes) starting at *address*. If *range* is given, the commands dump from the start to the end of *range*.

If neither *address* nor *range* is given, the commands assume the current dump address as the start of the range and the default size associated with the size of the object as the length of the range. Most Dump commands have a default range size of 128 bytes, but the Dump Real commands have a default range size of one real number.

The current dump address is the byte following the last byte specified in the previous Dump command. If no Dump command has been used during the session, the dump address is the start of the data segment (**DS**). For example, if you enter the Dump Words command with no argument as the first command of a session, the CodeView debugger displays the first 64 words (128 bytes) of data declared in the data segment. If you repeat the same command, the debugger displays the next 64 words following the ones dumped by the first command.

Note

Occasionally one of the Dump commands that display real numbers (Dump Short Reals, Dump Long Reals, or Dump 10-Byte Reals) will display a number containing one of the following character sequences: #NAN, #INF, or #IND. NAN (not a number) indicates that the data cannot be evaluated as a real number. INF (infinity) indicates that the data evaluate to infinity. IND (indefinite) indicates that the data evaluate to an indefinite number.

Sections 6.4.1–6.4.10 discuss the variations of the Dump commands in order of the size of data they display.

6.4.1 Dump

■ Syntax

D [[*address* | *range*]]

The Dump command displays the contents of memory at the specified *address* or in the specified *range* of addresses. The command dumps data in the format of the default type. The default type is the last type specified with a Dump, Enter, Watch Memory, or Tracepoint Memory command. If none of these commands has been entered during the session, the default type is bytes.

The Dump command displays one or more lines, depending on the address or range specified. Each line displays the address of the first item displayed. The Dump command must be separated by at least one space from any *address* or *range* value. For example, to dump memory starting at symbol a, use the command D a, not Da. The second syntax would be interpreted as the Dump ASCII command.

6.4.2 Dump Bytes

■ Syntax

DB [[*address* | *range*]]

The Dump Bytes command displays the hexadecimal and ASCII values of the bytes at the specified *address* or in the specified *range* of addresses. The command displays one or more lines, depending on the address or range supplied.

Each line displays the address of the first byte in the line, followed by up to 16 hexadecimal byte values. The byte values are immediately followed by the corresponding ASCII values. The hexadecimal values are separated by spaces, except the eighth and ninth values, which are separated by a dash (-). ASCII values are printed without separation. Unprintable ASCII values (less than 32 or greater than 126) are displayed as dots. No more than 16 hexadecimal values are displayed in a line. The command displays values and characters until the end of the *range* or, if no *range* is given, until the first 128 bytes have been displayed.

■ Example

```
>DB 0 36
3D5E:0000  53 6F 6D 65 20 6C 65 74-74 65 72 73 20 61 6E 64  Some letters and
3D5E:0010  20 6E 75 6D 62 65 72 73-3A 00 10 EA 89 FC FF EF   numbers:.......
3D5E:0020  00 F0 00 CA E4                                   .....
>
```

The example displays the byte values from DS:0 to DS:36 (DS:0x24). The data segment is assumed if no segment is given. ASCII characters are shown on the right.

6.4.3 Dump ASCII

■ Syntax

DA [[*address* | *range*]]

The Dump ASCII command displays the ASCII characters at a specified *address* or in a specified *range* of addresses. The command displays one or more lines of characters, depending on the *address* or *range* specified.

If no ending address is specified, the command dumps either 128 bytes or all bytes preceding the first null byte, whichever comes first. Up to 64 characters per line are displayed. Unprintable characters, such as carriage returns and line feeds, are displayed as dots. ASCII characters less than 32 and greater than 126 are unprintable.

■ Examples

```
>DA 0          ;*Example 1
3D7C:0000  Some letters and numbers:
>

>DA 0 36       ;*Example 2
3D7C:0000  Some letters and numbers:.....
>
```

Example 1 displays the ASCII values of the bytes starting at DS:0. Since no ending address is given, values are displayed up to the first null byte. In Example 2, an ending address is given, so the characters from DS:0 to DS:36 (DS:0x24) are shown. Unprintable characters are shown as dots.

6.4.4 Dump Integers

■ Syntax

DI [[*address* | *range*]]

The Dump Integers command displays the signed decimal values of the words (2-byte values) starting at *address* or in the specified *range* of addresses. The command displays one or more lines, depending on the address or range specified. Each line displays the address of the first integer in the line, followed by up to eight signed decimal words. The values are separated by spaces. The command displays values until the end of the *range* or until the first 64 integers have been displayed, whichever comes first.

Note

In the C language, the size of an integer is system dependent. In this manual an integer is a 2-byte value, since that is the size of integers produced with the Microsoft MS-DOS C Compiler.

■ Example

```
>DI 0 36
3D5E:0000    28499  25965  27680  29797  25972  29554  24864  25710
3D5E:0010    28192  28021  25954  29554     58  -5616   -887  -4097
3D5E:0020    -4096 -13824   2532
>
```

The example displays the byte values from `DS:0` to `DS:36` (`DS:0x24`). Compare the signed decimal numbers at the end of this dump with the same values shown as unsigned integers in Section 6.4.5.

6.4.5 Dump Unsigned Integers

■ Syntax

DU [[*address* | *range*]]

The Dump Unsigned Integers command displays the unsigned decimal values of the words (2-byte values) starting at *address* or in the specified *range* of addresses. The command displays one or more lines, depending on the address or range specified. Each line displays the address of the first unsigned integer in the line, followed by up to eight decimal words. The values are separated by spaces. The command displays values until the end of the *range* or until the first 64 unsigned integers have been displayed, whichever comes first.

■ Example

```
>DU 0 36
3D5E:0000    28499  25965  27680  29797  25972  29554  24864  25710
3D5E:0010    28192  28021  25954  29554     58  59920  64649  61439
3D5E:0020    61440  51712   2532
>
```

The example displays the byte values from DS:0 to DS:36 (DS:0x24). Compare the unsigned decimal numbers at the end of this dump with the same values shown as signed integers in Section 6.4.4.

6.4.6 Dump Words

■ Syntax

DW [[*address* | *range*]]

The Dump Words command displays the hexadecimal values of the words (2-byte values) starting at *address* or in the specified *range* of addresses. The command displays one or more lines, depending on the address or range specified. Each line displays the address of the first word in the line, followed by up to eight hexadecimal words. The hexadecimal values are separated by spaces. The command displays values until the end of the *range* or until the first 64 words have been displayed, whichever comes first.

■ Example

```
>DW 0 36
3D5E:0000  6F53 656D 6C20 7465 6574 7372 6120 646E
3D5E:0010  6E20 6D75 6562 7372 003A EA10 FC89 EFFF
3D5E:0020  F000 CA00 09E4
>
```

The example displays the word values from DS:0 to DS:36 (DS:0x24). No more than eight values per line are displayed.

6.4.7 Dump Double Words

■ Syntax

DD [[*address* | *range*]]

The Dump Double Words command displays the hexadecimal values of the double words (4-byte values) starting at *address* or in the specified *range* of addresses.

The command displays one or more lines, depending on the address or range specified. Each line displays the address of the first double word in the line, followed by up to four hexadecimal double-word values. The words of each double word are separated by a colon. The values are separated by spaces. The command displays values until the end of the *range* or until the first 32 double words have been displayed, whichever comes first.

■ Example

```
>DD 0 36
3D5E:0000  656D:6F53 7465:6C20 7372:6574 646E:6120
3D5E:0010  6D75:6E20 7372:6562 EA10:003A EFFF:FC89
3D5E:0020  CA00:F000 6F73:09E4
>
```

The example displays the double-word values from `DS:0` to `DS:36` (`DS:0x24`). No more than four double-word values per line are displayed.

6.4.8 Dump Short Reals

■ Syntax

DS [[*address* | *range*]]

The Dump Short Reals command displays the hexadecimal and decimal values of the short (4-byte) floating-point numbers at *address* or in the specified *range* of addresses.

The command displays one or more lines, depending on the address or range specified. Each line displays the address of the floating-point number in the first column. Next, the hexadecimal values of the bytes in the number are shown, followed by the decimal value of the number. The hexadecimal values are separated by spaces.

The decimal value has the following form:

[[**–**]]*digit.decimaldigits***E**+|–*mantissa*

If the number is negative, it will have a minus sign; positive numbers have no sign. The first digit of the number is followed by a decimal point. Six decimal places are shown following the decimal point. The letter **E** follows the decimal digits, and marks the start of a three-digit signed *mantissa*.

The command displays at least one value. If a *range* is specified, all values in the range are displayed.

■ Example

```
>DS s_pi
5E68:0100  DB OF 49 40  3.141593E+000
>
```

The example displays the short-real floating-point number at the address of the variable `s_pi`. Only one value is displayed per line.

6.4.9 Dump Long Reals

■ Syntax

DL [[*address* | *range*]]

The Dump Long Reals command displays the hexadecimal and decimal values of the long (8-byte) floating-point numbers at the specified *address* or in the specified *range* of addresses.

The command displays one or more lines, depending on the address or range specified. Each line displays the address of the floating-point number in the first column. Next, the hexadecimal values of the bytes in the number are shown, followed by the decimal value of the number. The hexadecimal values are separated by spaces.

The decimal value has the following form:

[[−]]*digit.decimaldigits***E**+|−*mantissa*

If the number is negative, it will have a minus sign; positive numbers have no sign. The first digit of the number is followed by a decimal point. Six decimal places are shown following the decimal point. The letter **E** follows the decimal digits, and marks the start of a three-digit signed *mantissa.*

The command displays at least one value. If a *range* is specified, all values in the range are displayed.

■ Example

```
>DL l_pi
5E68:0200  11 2D 44 54 FB 21 09 40  3.141593E+000
>
```

The example displays the long-real floating-point number at the address of the variable `l_pi`. Only one value per line is displayed.

6.4.10 Dump 10-Byte Reals

■ Syntax

DT [[*address* | *range*]]

The Dump 10-Byte Reals command displays the hexadecimal and decimal values of the 10-byte floating-point numbers at the specified *address* or in the specified *range* of addresses.

The command displays one or more lines, depending on the address or range specified. Each line displays the address of the floating-point number in the first column. Next, the hexadecimal values of the bytes in the number are shown, followed by the decimal value of the number. The hexadecimal values are separated by spaces.

The decimal value has the following form:

[[**–**]]*digit.decimaldigits***E+**|**–***mantissa*

If the number is negative, it will have a minus sign; positive numbers have no sign. The first digit of the number is followed by a decimal point. Six decimal places are shown following the decimal point. The letter **E** follows the decimal digits, and marks the start of a three-digit signed *mantissa*.

The command displays at least one value. If a *range* is specified, all values in the range are displayed.

■ Example

```
>DT t_pi
5E68:0300  DE 87 68 21 A2 DA 0F C9 00 40  3.141593E+000
>
```

The example displays the 10-byte-real floating-point number at the address of the variable `t_pi`. Only one number per line is displayed.

6.5 Register Command

The Register command has two functions. It displays the contents of the central processing unit (CPU) registers. It can also change the values of the registers. The display features of the Register command are explained here. The modification features of the command are explained in Chapter 10, "Modifying Code or Data."

■ Mouse

To display the registers with the mouse, point to Options on the menu bar, press a mouse button and drag the highlight down to the Registers selection, then release the button. The register window will appear on the right side of the screen. If the register window is already on the screen, the same command removes it.

■ Keyboard

To display the registers using a keyboard command in window mode, press the F2 key. The register window will appear on the right side of the screen. If the register window is already on the screen, the same command will remove it.

In sequential mode, the F2 key will display the current status of the registers. (This produces the same effect as entering the Register dialog command with no argument.)

■ Dialog

To display the registers in the dialog window (or sequentially in sequential mode), enter a command line with the following syntax:

R

The current values of all registers and flags are displayed (in the dialog window in window mode). The instruction at the address pointed to by the current **CS** and **IP** register values is also shown. (The Register command can also be given with arguments, but only when used to modify registers, as explained in Chapter 10, "Modifying Code or Data.")

If the display mode is source (**S+**) or mixed (**S&**) (see Chapter 9, "Examining Code," for more information), the current source line is also displayed by the Register command. If an operand of the instruction contains memory expressions or immediate data, the CodeView debugger will evaluate operands and show the value to the right of the instruction. If the **CS** and **IP** registers are currently at a breakpoint location, the register display will indicate the breakpoint number.

In sequential mode, the Trace (**T**), Program Step (**P**), and Go (**G**) commands show registers in the same format as the Register command.

■ Examples

```
>S&        ;* Example 1
mixed
>R
AX=0043  BX=0043  CX=025C  DX=0000  SP=18F4  BP=18F8  SI=04BA  DI=1946
DS=5BF2  ES=5BF2  SS=5BF2  CS=5701  IP=026E  NV UP EI PL NZ NA PO NC
92:                    ++letters;

5701:026E FF067201      INC      Word Ptr [_letters (0172)]     DS:0172=0000
>
```

Example 1 displays all register and flag values, as well as the instruction at the address pointed to by the **CS** and **IP** registers. Since the mode has been set to mixed (**S&**), the current source line is also shown.

The memory operand `[_letters]` in the instruction is evaluated on the right side of the screen as `DS:0172=0000`. This means that the variable `letters` (at offset **0x0172** of the data segment) currently has a value of 0. The next instruction (`INC Word Ptr [_letters]`) will increment the value.

```
>S-        ;* Example 2
source
>R
AX=024F  BX=0001  CX=0000  DX=0000  SP=1900  BP=1908  SI=04BA  DI=1946
DS=5BF2  ES=5BF2  SS=5BF2  CS=5701  IP=021F  NV UP EI PL NZ NA PO NC
5701:021F 807EFC20       CMP       Byte Ptr [code],20          ;BR2
>
```

In Example 2, the display mode is set to assembly (**S–**), so no source line is shown. Note the breakpoint number at the right of the last line, indicating that the current address is at breakpoint 2.

6.6 8087 Command

The 8087 command dumps the contents of the 8087 registers. This command is only useful if you have an 8087 or 80287 coprocessor chip on your system.

Note

This section does not attempt to explain how the registers of the Intel® 8087 and 80287 processors are organized or how they work. In order to interpret the command output, you must learn about the chip from an Intel reference manual or other book on the subject.

■ Mouse

The 8087 command cannot be executed with the mouse.

■ Keyboard

The 8087 command cannot be executed with a keyboard command.

■ Dialog

To display the status of the 8087 or 80287 chip using a dialog command, enter a command line with the following syntax:

7

The current status of the chip is output when you enter the command. In window mode, the output is to the dialog window. If you do not have an 8087 or 80287 chip, all registers in the output will contain 0.

■ Example

```
>7
Control 037F  (Projective closure, Round nearest, 64-bit precision)
                        iem=0 pm=1 um=1 om=1 zm=1 dm=1 im=1
Status  6004  cond=1000 top=4 pe=0 ue=0 oe=0 ze=1 de=0 ie=0
Tag     A1FF  instruction=59380  operand=59360 . opcode=D9EE
Stack         Exp  Mantissa             Value
ST(3) special 7FFF 8000000000000000 = + Infinity
ST(2) special 7FFF 0101010101010101 = + Not a Number
ST(1) valid   4000 C90FDAA22168C235 = +3.14159226511039OE+000
ST(0) zero    0000 0000000000000000 = +0.0000000000000000E+000
>
```

In the example above, the first line of the dump shows the current closure method, rounding method, and precision. The number `037F` is the hexadecimal value in the control register. The rest of the line interprets the bits of the number. The closure method can be either projective (as in the example) or affine. The rounding method can be either rounding to the nearest even number (as in the example), rounding down, rounding up, or the chop method of rounding (truncating toward zero). The precision may be 64 bits (as in the example), 53 bits, or 24 bits.

The second line of the display indicates whether each exception mask bit is set or cleared. The masks are: interrupt-enable mask (`iem`), precision mask (`pm`), underflow mask (`um`), overflow mask (`om`), zero-divide mask (`zm`), denormalized-operand mask (`dm`), and invalid-operation mask (`im`).

The third line of the display shows the hexadecimal value of the status register (`6004` in the example), then interprets the bits of the register. The condition code (`cond`) in the example is the binary number `1000`. The top of the stack (`top`) is register 4 (shown in decimal). The other bits shown are: precision exception (`pe`), underflow exception (`ue`), overflow exception (`oe`), zero-divide exception (`ze`), denormalized-operand exception (`de`), and invalid-operation exception (`ie`).

The fourth line of the display first shows the hexadecimal value of the tag register (A1FF in the example). It then gives the hexadecimal values of the instruction (59380), the operand (59360), and the operation code, or opcode, (D9EE).

The fifth line is a heading for the subsequent lines, which contain the contents of each 8087 or 80287 stack register. The registers in the example contain four types of numbers that may be held in these registers. Starting from the bottom, register 0 contains zero. Register 1 contains a valid real number. Its exponent (in hexadecimal) is 4000 and its mantissa is C90FDAA22168C235. The number is shown in scientific notation in the rightmost column. Register 2 contains a value that cannot be interpreted as a number, and register 3 contains infinity.

Chapter 7
Managing Breakpoints

7.1 Introduction 121
7.2 Breakpoint Set Command 121
7.3 Breakpoint Clear Command 124
7.4 Breakpoint Disable Command 125
7.5 Breakpoint Enable Command 127
7.6 Breakpoint List Command 128

7.1 Introduction

The CodeView debugger enables you to control program execution by setting breakpoints. A breakpoint is an address that stops program execution each time the address is encountered. By setting breakpoints at key addresses in your program, you can "freeze" program execution and examine the status of memory or expressions at that point.

The commands listed below control breakpoints:

Command	Action
Breakpoint Set (**BP**)	Sets a breakpoint and optionally a pass count and break commands
Breakpoint Clear (**BC**)	Clears one or more breakpoints
Breakpoint Disable (**BD**)	Disables one or more breakpoints
Breakpoint Enable (**BE**)	Enables one or more breakpoints
Breakpoint List (**BL**)	Lists all breakpoints

In addition to these commands, the Watchpoint (**WP**) and Tracepoint (**TP**) commands can be used to set conditional breakpoints. These commands are explained in Chapter 8, "Managing Watch Statements." The Breakpoint commands are discussed in sections 7.2–7.6.

7.2 Breakpoint Set Command

The Breakpoint Set command creates a breakpoint at a specified address. Any time a breakpoint is encountered during program execution, the program halts and waits for a new command.

The CodeView debugger allows up to 20 breakpoints (0 through 19). Each new breakpoint is assigned the next available number. Breakpoints remain in memory until you delete them (see Section 6.3 for more information) or until you quit the debugger. They are not canceled when you restart the program. This enables you to set up a complicated series of breakpoints, then execute through the program several times without resetting the breakpoints.

If you try to set a breakpoint at a comment line or other source line that does not correspond to code, the CodeView debugger displays the following message:

```
No code at this line number
```

■ Mouse

To set a breakpoint with the mouse, point to the source line or instruction where you want to set the breakpoint, then click the left button. The line will be displayed in high-intensity text, and will remain so until you remove or disable the breakpoint.

■ Keyboard

To set a breakpoint with a keyboard command in window mode, move the cursor to the source line or instruction where you want to set a breakpoint. You may have to press the F6 key to move the cursor to the display window. When the cursor is on the appropriate source line, press the F9 key. The line will be displayed in high-intensity text, and will remain so until you remove the breakpoint.

In sequential mode, the F9 key can be used to set a breakpoint at the current location. You must use the dialog version of the command to set a breakpoint at any other location.

■ Dialog

To set a breakpoint using a dialog command, enter a command line with the following syntax:

BP [[*address* [[*passcount*]] [[" *commands*"]]]

If no *address* is given, a breakpoint is created on the current source line in source mode, or on the current instruction in assembly mode. You can specify the *address* either in the *segment*:*offset* format, or as a source line, a function name, or a label. If you give an offset address, the code segment is assumed.

The dialog version of the command is more powerful than the mouse or keyboard version in that it allows you to give a *passcount* and a string of

commands. The *passcount* specifies the first time the breakpoint is to be taken. For example, if the pass count is 5, the breakpoint will be ignored the first four times it is encountered, and taken the fifth time.

The *commands* are a list of dialog commands enclosed in quotation marks (" ") and separated by semicolons (;). For example, if you specify the commands as "`? code;T`", the CodeView debugger will automatically display the value of the variable `code` and then execute the Trace command each time the breakpoint is encountered. The Trace and Display Expression commands are described in Chapter 6, "Examining Data and Expressions," and Chapter 5, "Executing Code," respectively.

In window mode, a breakpoint entered with a dialog command has exactly the same effect as one created with a window command. The source line or instruction corresponding to the breakpoint location is shown in high-intensity text.

In sequential mode, information about the current instruction will be displayed each time you execute to a breakpoint. The register values, the current instruction, and the source line may be shown, depending on the display mode. See Chapter 9, "Examining Code," for more information about display modes.

When a breakpoint address is shown in the assembly-language format, the breakpoint number will be shown as a comment to the right of the instruction. This comment appears even if the breakpoint is disabled (but not if it is deleted).

■ Examples

```
>BP .19                          ;*Example 1
>

>BP display 10 "?++counter;G"  ;*Example 2
>

>S-                              ;*Example 3
>BP 0x205
>G
AX=011D  BX=0183  CX=0000  DX=0000  SP=12FE  BP=1306  SI=04BA  DI=1344
DS=5E61  ES=5E61  SS=5E61  CS=596D  IP=0205  NV UP EI PL NZ NA PO NC
596D:0205 E97800          JMP       _countwords+9a (0280)     ;BR1
>
```

Example 1 creates a breakpoint at line 19 of the current source file (or if there is no executable statement at line 19, at the first executable statement after line 19).

Example 2 creates a breakpoint at the address of the function `display`. The breakpoint is passed over nine times before being taken on the 10th pass. Each time execution stops for the breakpoint, the quoted commands are executed. The Display Expression command increments `counter`, then the Go command restarts execution. If `counter` is set to 0 when the breakpoint is set, this has the effect of counting the number of times the breakpoint address is passed.

Example 3, shown in sequential mode, first sets the mode to assembly, then creates a breakpoint at the offset address `0xFA2` in the default (**CS**) segment. The Go command (**G**) is then used to execute to the breakpoint. Note that in the output to the Go command, the breakpoint number is shown as an assembly-language comment (`;BR1`) to the right of the current instruction.

7.3 Breakpoint Clear Command

The Breakpoint Clear command permanently removes one or more previously set breakpoints.

■ Mouse

To clear a single breakpoint with the mouse, point to the breakpoint line or instruction you want to clear. Breakpoint lines are shown in high-intensity text. Press the left mouse button. The line will be shown in normal text to indicate that the breakpoint has been removed.

To remove all breakpoints with the mouse, point to Run on the menu bar, press a mouse button and drag the highlight down to the Clear Breakpoints selection, then release the button.

■ Keyboard

To clear a single breakpoint with a keyboard command, move the cursor to the breakpoint line or instruction you want to clear. Breakpoint lines are shown in high-intensity text. Press the F9 key. The line will be shown in normal text to indicate that the breakpoint has been removed.

To remove all breakpoints using a keyboard command, press ALT-R to open the Run menu, then press ALT-C to select Clear Breakpoints.

■ Dialog

To clear breakpoints using a dialog command, enter a command line with the following syntax:

BC *list*
BC *

If *list* is specified, the command removes the breakpoints named in the list. The *list* can be any combination of integer values from 0 to 19. You can use the Breakpoint List command (**BL**) if you need to see the numbers for each existing breakpoint. If an asterisk (*) is given as the argument, all breakpoints are removed.

■ Examples

```
>BC 0 4 8          ;*Example 1
>

>BC *              ;*Example 2
>
```

Example 1 removes breakpoints 0, 4, and 8. Example 2 removes all breakpoints.

7.4 Breakpoint Disable Command

The Breakpoint Disable command temporarily disables one or more existing breakpoints. The breakpoints are not deleted. They can be restored at any time using the Breakpoint Enable command (**BE**).

When a breakpoint is disabled in window mode, it is shown in the display window with normal text. When it is enabled, it is shown in high-intensity text.

Note

All disabled breakpoints are automatically enabled whenever you restart the program being debugged. The program can be restarted with the Start or Restart selections from the Run menu, or with the Restart dialog command (**L**). See Chapter 11, "Using System-Control Commands."

■ Mouse

The Breakpoint Disable command cannot be executed with the mouse.

■ Keyboard

The Breakpoint Disable command cannot be executed with a keyboard command.

■ Dialog

To disable breakpoints using a dialog command, enter a command line with the following syntax:

BD *list*
BD *

If *list* is specified, the command disables the breakpoints named in the list. The *list* can be any combination of integer values from 0 to 19. Use the Breakpoint List command (**BL**) if you need to see the numbers for each existing breakpoint. If an asterisk (*) is given as the argument, all breakpoints are disabled.

The window commands for setting and clearing breakpoints can also be used to enable or clear disabled breakpoints.

■ Examples

```
>BD 0 4 8          ;*Example 1
>

>BD *              ;*Example 2
>
```

Example 1 disables breakpoints 0, 4, and 8. Example 2 disables all breakpoints.

7.5 Breakpoint Enable Command

The Breakpoint Enable command enables breakpoints that have been temporarily disabled with the Breakpoint Disable command.

■ Mouse

To enable a disabled breakpoint with the mouse, point to the source line or instruction of the breakpoint, then click the left button. The line will be displayed in high-intensity text, and will remain so until you remove or disable the breakpoint. This is the same as creating a new breakpoint at that location.

■ Keyboard

To enable a disabled breakpoint using a keyboard command, move the cursor to the source line or instruction of the breakpoint, then press the F9 key. The line will be displayed in high-intensity text, and will remain so until you remove or disable the breakpoint. This is the same as creating a new breakpoint at that location.

■ Dialog

To enable breakpoints using a dialog command, enter a command line with the following syntax:

BE *list*
BE *

If *list* is specified, the command enables the breakpoints named in the list. The *list* can be any combination of integer values from 0 to 19. Use the Breakpoint List command (**BL**) if you need to see the numbers for each existing breakpoint. If an asterisk (*) is given as the argument, all breakpoints are enabled. The CodeView debugger ignores all or part of the command if you try to enable a breakpoint that is not disabled.

■ Examples

```
>BE 0 4 8          ;*Example 1
>

>BE*               ;*Example 2
>
```

Example 1 enables breakpoints 0, 4, and 8. Example 2 enables all disabled breakpoints.

7.6 Breakpoint List Command

The Breakpoint List command lists current information about all breakpoints.

■ Mouse

The Breakpoint List command cannot be executed with the mouse.

■ Keyboard

The Breakpoint List command cannot be executed with a keyboard command.

■ Dialog

To list breakpoints using a dialog command, enter a command line with the following syntax:

BL

The command displays the breakpoint number, the enabled status, the address, the function, and the line number. If the breakpoint does not fall on a line number, an offset is shown from the nearest previous line number. The pass count and break commands are shown if they have been set. The status can be `e` for enabled, `d` for disabled. If no breakpoints are currently defined, nothing is displayed.

■ Example

```
>BL
0 e 56C4:0105  _arctan:10
1 d 56C4:011E  _arctan:19                    (pass = 10) "T;T"
2 e 56C4:00FD  _arctan:9+6
>
```

In the example, breakpoint 0 is enabled at address `56C4:0105`. This address is in function `arctan` and is at line `10` of the current source file. No pass count or break commands have been set.

Breakpoint 1 is currently disabled, as indicated by the `d` after the breakpoint number. It also has a pass count of 10, meaning that the breakpoint will not be taken until the 10th time it is encountered. The command string at the end of the line indicates that each time the breakpoint is taken, the Trace command will automatically be executed twice.

The line number for breakpoint 2 has an offset. The address is 0x6 bytes beyond the address for line number 9 in the current source file. This indicates that the breakpoint was probably set in assembly mode, since it would be difficult to set a breakpoint anywhere except on a source line in source mode.

Chapter 8
Managing Watch Statements

8.1 Introduction 133

8.2 Setting Watch-Expression and Watch-Memory Statements 134

8.3 Setting Watchpoints 138

8.4 Setting Tracepoints 141

8.5 Deleting Watch Statements 146

8.6 Listing Watchpoints and Tracepoints 148

8.1 Introduction

Watch Statement commands are among the Microsoft CodeView debugger's most powerful features. They enable you to set, delete, and list watch statements. Watch statements are specifications that describe expressions or areas of memory to watch. Some watch statements also specify conditional breakpoints that may or may not be taken, depending on the value of the expression or memory area.

The Watch Statement commands are summarized below:

Command	Action
Watch (**W**)	Sets an expression or range of memory to be watched
Watchpoint (**WP**)	Sets a conditional breakpoint that will be taken when the expression becomes nonzero (true)
Tracepoint (**TP**)	Sets a conditional breakpoint that will be taken when a given expression or range of memory changes
Watch Delete (**Y**)	Deletes one or more watch statements
Watch List (**W**)	Lists current watch statements

Watch statements are like breakpoints in that they remain in memory until you specifically remove them or quit the CodeView debugger. They are not canceled when you restart the program being debugged. This enables you to set a complicated series of watch statements, then execute through the program several times without resetting the watch statements.

In window mode, Watch Statement commands can be entered either in the dialog window or with menu selections. Current watch statements are shown in a watch window that appears between the menu bar and the source window.

In sequential mode, the Watch, Tracepoint, and Watchpoint commands can be used, but since there is no watch window, you cannot see the watch statements and their values. You must use the Watch List command to examine the current watch statements.

Note

In order to set a watch statement containing a local variable, you must be in the function where the variable is defined. If the current line is not in the function, the CodeView debugger displays the message UNKNOWN SYMBOL. When you exit from a function containing a local variable referenced in a watch statement, the value of the statement is displayed as UNKNOWN SYMBOL. When you reenter the function, the local variable will again have a value. You can avoid this limitation by using the period operator to specify both the function and the variable. For example, enter `main.argc` instead of just `argc`.

8.2 Setting Watch-Expression and Watch-Memory Statements

The Watch command is used to set a watch statement that describes an expression or a range of addresses in memory. The value or values described by this watch statement are shown in the watch window. The watch window is updated to show new values each time the value of the watch statement changes during program execution. Since the watch window does not exist in sequential mode, you must use the Watch List command to examine the values of watch statements.

When setting a watch expression, you can specify the format in which the value will be displayed. Type the expression followed by a comma and a type specifier. If you do not give a type specifier, the CodeView debugger displays the value in a default format. See Chapter 6, "Examining Data and Expressions," for more information about type specifiers and the default format.

Note

If your program directly accesses absolute addresses used by IBM or IBM-compatible computers, you may sometimes get unexpected results with the Display Expression and Dump commands. However, the Watch command will usually show the correct values. This problem can arise if the CodeView debugger and your program try to use the same memory location.

This often occurs when a program reads data directly from the screen buffer of the display adapter. If you have an array called `screen` that is initialized to the starting address of the screen buffer, the command `DB screen L 16` will display data from the CodeView display rather than from the display of the program you are debugging. The command `WB screen L 16` will display data from the program's display (provided screen swapping or screen flipping was specified at start-up). This happens because watch-statement values are updated during program execution, and any values read from the screen buffer will be taken from the output screen rather than from the debugging screen.

■ Mouse

To set a watch-expression statement using the mouse, point to Watch on the menu bar, press a mouse button and drag the highlight down to the Add Watch selection, then release the button. A dialog box appears, asking for the expression to be watched. Type the expression and press the ENTER key or a mouse button.

You cannot use the mouse version of the command to specify a range of memory to be watched, as you can with the dialog version.

■ Keyboard

To set a watch-expression statement with a keyboard command, press ALT-W to open the Watch menu, then press ALT-A to select Add Watch. A dialog box appears, asking for the expression to be watched. Type the expression and press the ENTER key.

You cannot use the keyboard version of the command to specify a range of memory to be watched, as you can with the dialog version.

■ Dialog

To set a watch-expression statement or watch-memory statement using a dialog command, enter a command line with the following syntax:

W? *expression*[[,*format*]]	Watch expression
W[[*type*]] *range*	Watch memory

An *expression* used with the Watch command can be a simple variable, or a complex expression using several variables and operators. The expression should be no longer than the width of the watch window. You can specify *format* using a **printf** type specifier. See Chapter 6, "Examining Data and Expressions," or Appendix A, "Command and Mode Summary," for more information.

When watching a memory location, *type* is a one-letter size specifier from the following list:

Specifier	Size
None	Default type
B	Byte
A	ASCII
I	Integer (signed decimal word)
U	Unsigned (unsigned decimal word)
W	Word
D	Double word
S	Short real
L	Long real
T	10-byte real

The default type used if no type size is specified is the last type used by a Dump, Enter, Watch Memory, or Tracepoint Memory command. If none of these commands has been used during the session, the default type is byte.

The data will be displayed in a format similar to that used by the Dump commands (see Chapter 6, "Examining Data and Expressions," for more information). The *range* can be any length, but only one line of data will be displayed in the watch window. If you do not specify an ending address for the range, the default range is one object.

■ Examples

The following dialog commands display three watch statements in the watch window:

```
W? code,c                          ;* Example 1
W? (float)letters/words,f          ;* Example 2
WB buffer L 7                      ;* Example 3
```

These commands produce the watch window in Figure 8.1.

```
File  Search  View  Run  Watch  Options  Calls  Trace!  Go!          count.exe
                                  count.c
0) code,c  :  I
1) (float)letters/words,f  :  4.777778
2) 3F65:0B20  20 20 43 4F 55 4E 54   COUNT

89:                     if (!inword) {
90:                             if (code > ' ') {
91:                                     analyze(code,inword);
92:                                     inword = TRUE;
93:                                     ++words;
94:                                     ++characters;
95:                             }
96:                     } else {
97:                             if (code <= ' ')
98:                                     inword = FALSE;
99:                             else {
100:                                    ++characters;
101:                                    analyze(code,inword);
102:                            }

>W? code,c                        ;* Example 1
>W? (float)letters/words,f        ;* Example 2
>WB buffer L 7                    ;* Example 3
>
```

Figure 8.1 Watch-Command Statements in the Watch Window

Example 1 displays the current value of the variable `code`. The `c` type specifier indicates that the value is to be displayed as a character.

Example 2 displays the value of the expression `(float)letters/words`. The type cast is necessary to specify real-number division rather than integer division. The expression will be displayed in the default floating-point format.

Example 3 displays the first seven values at the address of the variable `buffer`. The characters to the right of the byte values in the watch window represent the corresponding ASCII characters (the word `COUNT` in the example). Unprintable values (less than 33 or greater than 126) are represented by dots. ASCII values are only shown if the specified *type* is **B** (byte) or **A** (ASCII).

8.3 Setting Watchpoints

The Watchpoint command is used to set a conditional breakpoint called a watchpoint. A watchpoint breaks program execution when the expression described by its watch statement becomes true. You can think of watchpoints as "break when" points, since the break occurs when the specified expression becomes true (nonzero).

A watch statement created by the Watchpoint command describes the expression that will be watched and compared to 0. The statement remains in memory until you delete it or quit the CodeView debugger. Any valid CodeView expression can be used as the watchpoint expression as long as the expression is not wider than the watch window.

In window mode, watchpoint statements and their values are displayed in high-intensity text in the watch window. In sequential mode, there is no watch window, so the values of watchpoint statements can only be displayed with the Watch List command (see Section 8.6 for more information).

Although a watchpoint expression can be any valid CodeView expression, the command works best with relational expressions that use the operators ==, <=, >=, !=, <, and >. Relational expressions always evaluate to 0 (false) or 1 (true). Other expressions usually evaluate to larger numbers, thus the break is always taken.

Note

An example of the problems involved with using nonrelational expressions occurs if you accidentally type the simple assignment operator (=) when you mean to use the equality operator (==). Using the simple assignment operator is legal, but it has unexpected side effects. For example, the expression `count=6` resets the value of `count` to `6` every time you try to execute code. The effect is that the break is always taken, so code is always executed one line at a time.

■ Mouse

To set a watchpoint statement with the mouse, point to Watch on the menu bar, press a mouse button and drag the highlight down to the Watchpoint selection, then release the button. A dialog box appears, asking for the expression to be watched. Type the expression and press the ENTER key or a mouse button.

■ Keyboard

To execute the Watchpoint command with a keyboard command, press ALT-W to open the Watch menu, then press ALT-W to select Watchpoint. A dialog box appears, asking for the expression to be watched. Type the expression and press the ENTER key.

■ Dialog

To set a watchpoint using a dialog command, enter a command line with the following syntax:

WP? *expression*[[,*format*]]

The *expression* can be any valid CodeView expression (usually a relational expression). You can enter *format* as a **printf** type specifier, but there is little reason to do so, since the expression value is normally either 1 or 0.

■ Examples

The following dialog commands display two watch statements in the watch window:

```
WP? lines==11                ;* Example 1
WP? buffer[count]<65         ;* Example 2
```

These commands produce the watch window in Figure 8.2.

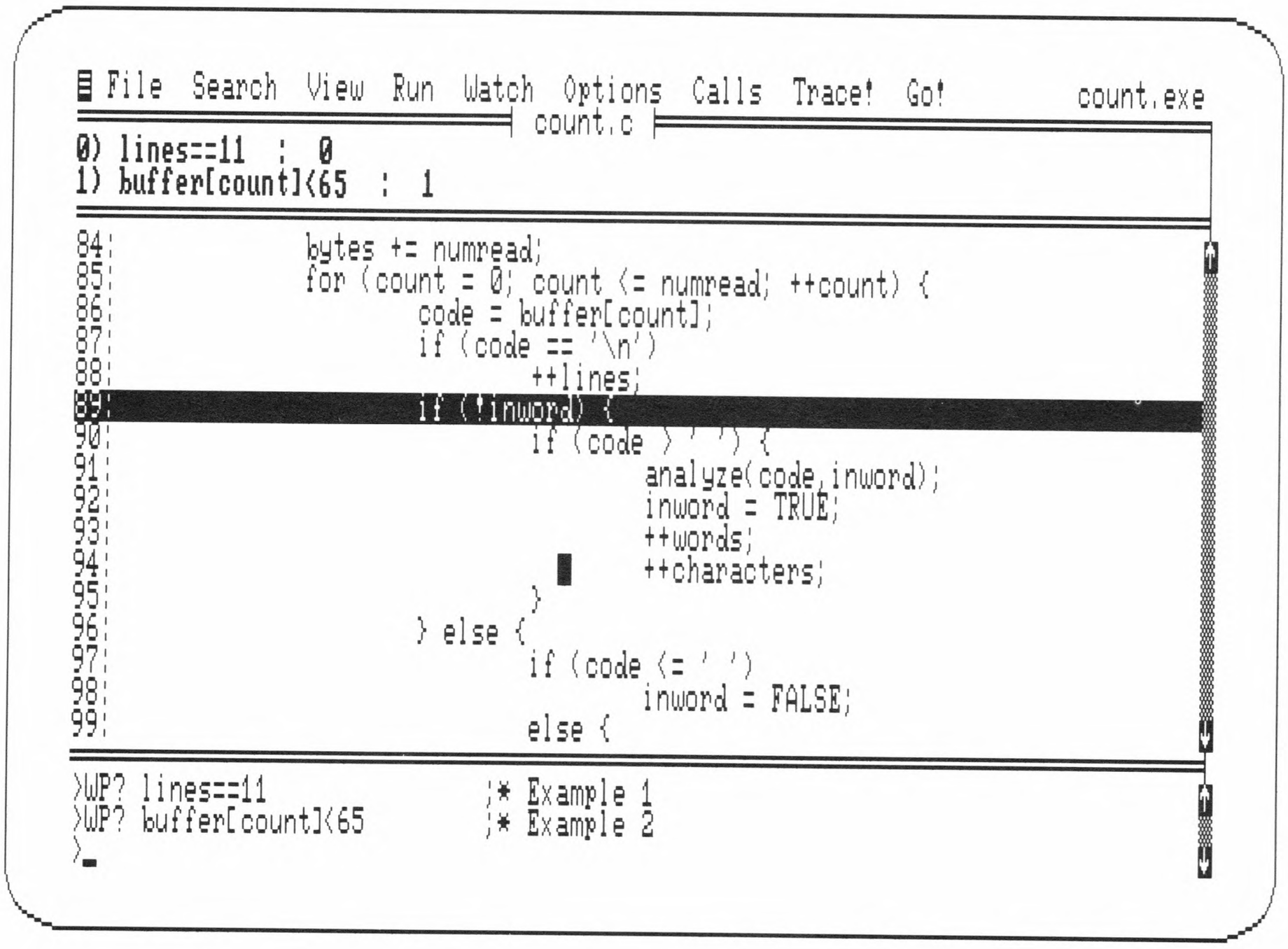


Figure 8.2 Watchpoint-Command Statements in the Watch Window

Example 1 instructs the CodeView debugger to break when the variable `lines` is equal to 11. After setting this watchpoint, you could use the Go command to execute until the condition becomes true.

Example 2 instructs the CodeView debugger to break when the expression `buffer[count]` becomes less than 65 (ASCII A). If `buffer` is an array of ASCII characters and `count` is a loop counter that points to the current position in the array, then the watchpoint has the effect of breaking for digits, most punctuation marks, and other characters less than 65.

Note

Setting watchpoints significantly slows execution of the program being debugged. The CodeView debugger has to check to see if the expression is true each time a source line is executed in source mode, or each time an instruction is executed in assembly mode. Be careful when setting watchpoints near large or nested loops. A loop that executes almost instantly when run from MS-DOS can take many minutes if executed from within the debugger with several watchpoints set.

Tracepoints do not slow CodeView execution as much as watchpoints, so you should use tracepoints when possible. For example, although you can set a watchpoint on a Boolean variable (`WP? moving`), a tracepoint on the same variable (`TP? moving`) has essentially the same effect and does not slow execution as much.

If you enter a seemingly endless loop, press CONTROL-BREAK or CONTROL-C to exit. You will soon learn the size of loop you can safely execute when watchpoints are set.

8.4 Setting Tracepoints

The Tracepoint command is used to set a conditional breakpoint called a tracepoint. A tracepoint breaks program execution when there is a change in the value of a specified expression or range of memory.

The watch statement created by the Tracepoint command describes the expression or memory range to be watched and tested for change. The statement remains in memory until you delete it or quit the CodeView debugger.

In window mode, tracepoint statements and their values are shown in high-intensity text in the watch window. In sequential mode, there is no watch window, so the values of tracepoint statements can only be displayed with the Watch List command (see Section 8.6, "Listing Watchpoints and Tracepoints," for more information).

An expression used with the Tracepoint command must evaluate to an "lvalue". In other words, the expression must refer to a memory location of not more than 128 bytes. For example, `i==10` would be invalid because it is either 1 (true) or 0 (false) rather than a value stored in memory. The expression `sym1+sym2` is invalid because it is the calculated sum of the value of two memory locations. The expression `buffer` would be invalid if `buffer` is an array of 130 bytes, but valid if the array is 120 bytes. Note that if buffer is declared as `char buffer[63]`, the Tracepoint command given with the expression `buffer` checks all 64 bytes of the array. The same command given with the expression `buffer[32]` means that only one byte (the 33rd) will be checked.

Note

Register variables are not considered lvalues. Therefore, if `i` is declared as `register int i`, the command `TP? i` is invalid. However, you can still check for changes in the value of `i`. Use the Examine Symbols command to learn which register contains the value of `i`. Then learn the value of `i`. Finally, set up a watchpoint to test the value. For example, use the following sequence of commands:

```
>X? i
3A79:0264 int                   div()
            SI          int                 i
>?i
10
>WP? @SI!=10
>
```

When setting a tracepoint expression, you can specify the format in which the value will be displayed. Type the expression followed by a comma and a type specifier. If you do not give a type specifier, the CodeView debugger displays the value in a default format. See Chapter 6, "Examining Data and Expressions," for more information about type specifiers and the default format.

■ Mouse

To set a tracepoint-expression statement with the mouse, point to Watch on the menu bar, press a mouse button and drag the highlight down to the Tracepoint selection, then release the button. A dialog box appears, asking for the expression to be watched. Type the expression and press the ENTER key or a mouse button.

You cannot specify a range of memory to be watched with the mouse version of the command as you can with the dialog version.

■ Keyboard

To set a tracepoint-expression statement with a keyboard command, press ALT-W to open the Watch menu, then press ALT-T to select Tracepoint. A dialog box appears, asking for the expression to be watched. Type the expression and press the ENTER key.

You cannot use the keyboard version of the command to specify a range of memory to be watched, as you can with the dialog version.

■ Dialog

To set a tracepoint using a dialog command, enter a command line with the following syntax:

TP? *expression*,[[*format*]]	Expression tracepoint
TP[[*type*]] *range*	Memory tracepoint

An *expression* used with the Tracepoint command can be a simple variable or a complex expression using several variables and operators. The expression should not be longer than the width of the watch window. You can specify *format* using a **printf** type specifier if you do not want the value to be displayed in the default format (decimal for integers or floating point for real numbers). See Section 6.2, "Display Expression Command," for more information.

In the memory-tracepoint form, *range* must be a valid address range and *type* must be a one-letter memory-size specifier. If you specify only the start of the range, the Codeview debugger displays one object as the default.

Although no more than one line of data will be displayed in the watch window, the range to be checked for change can be any size up to 128 bytes. The data will be displayed in the format used by the Dump commands (see Chapter 6, "Examining Data and Expressions," for more information). The valid memory-size specifiers are listed below:

Specifier	Size
None	Default type
B	Byte
A	ASCII
I	Integer (signed decimal word)
U	Unsigned (unsigned decimal word)
W	Word
D	Double word
S	Short real
L	Long real
T	10-byte real

The default type if no type size is specified is the last type used by a Dump, Enter, Watch Memory, or Tracepoint Memory command. If none of these commands has been used during the session, the default type is byte.

■ Examples

The following dialog commands display three watch statements in the watch window:

```
TP? prime        ;* Example 1
TPB flags[0] L 16  ;* Example 2
```

These commands produce the watch window in Figure 8.3 on the following page.

```
File  Search  View  Run  Watch  Options  Calls  Trace!  Go!          sieve.exe
                                  sieve.c
0) prime  :  3
1) 3AF5:0830  01 01 01 00 01 01 00 01 01 01 01 01 01 01 01 01 ................

19:       for (iter = 1; iter <= 10; iter++) {    /* do program 10 times */
20:            count = 0;                  /* initialize prime counter */
21:            for (i = 0; i <= SIZE; i++)   /* set all flags true */
22:                 flags[i] = TRUE;
23:            for (i = 0; i <= SIZE; i++) {
24:                 if (flags[i]) {              /* found a prime */
25:                      prime = i + i + 3;  /* twice index + 3 */
26:                      for (k = i + prime; k <= SIZE; k += prime)
27:                           flags[k] = FALSE;   /* kill all multiples */
28:                      count++;       /* primes found */
29:                 }
30:            }
31:       }
32:       printf("%d primes.\n",count);      /*primes found in 10th pass */
33:       printf("sieve.c finished\n");
34:       }

>TP? prime              ;* Example 1
>TPB flags[0] L 16      ;* Example 2
>
```

Figure 8.3 Tracepoints in the Watch Window

Example 1 instructs the CodeView debugger to stop whenever the value of the variable `prime` changes.

Example 2 instructs the debugger to stop whenever there is a change in the value of any of the 16 bytes starting at the address of the variable `flags`.

Note

Setting tracepoints significantly slows execution of the program being debugged. The CodeView debugger has to check to see if the expression or memory range has changed each time a source line is executed in source mode or each time an instruction is executed in assembly mode. However, tracepoints do not slow execution as much as watchpoints. Be careful when setting tracepoints near large or nested loops. A loop that executes almost instantly when run from MS-DOS can take many minutes if executed from within the debugger with several tracepoints set.

If you enter a seemingly endless loop, press CONTROL-BREAK or CONTROL-C to exit. Often you can tell how far you went in the loop by the value of the tracepoint when you exited.

8.5 Deleting Watch Statements

The Watch Delete command enables you to delete watch statements that were set previously with the Watch, Watchpoint, or Tracepoint command.

When you delete a watch statement in window mode, the statement disappears and the watch window closes around it. For example, if there are three watch statements in the window and you delete statement 1, the window is redrawn with one less line. Statement 0 remains unchanged, but statement 2 becomes statement 1. If there is only one statement, the window disappears.

■ Mouse

To delete a watch statement with the mouse, point to Watch on the menu bar, press a mouse button and drag the highlight down to the Delete Watch selection, then release the button. A dialog box appears, containing all the watch statements. Point to the statement you want to delete and press the ENTER key or a mouse button. The dialog box disappears and the watch window is redrawn without the deleted watch statement.

■ Keyboard

To execute the Execute command with a keyboard command, press ALT-W to open the Watch menu, then press ALT-D to select Delete Watch. A dialog box appears, containing all the watch statements. Use the UP and DOWN ARROW keys to move the cursor to the statement you want to delete, then press the ENTER key. The dialog box disappears and the watch window is redrawn without the deleted watch statement.

■ Dialog

To delete watch statements using a dialog command, enter a command line with the following syntax:

Y *number*

When you set a watch statement, it is automatically assigned a number (starting with 0). In window mode, the number appears to the left of the watch statement in the watch window. In sequential mode, you can use the Watch List (**W**) command to view the numbers of current watch statements.

You can delete existing watch statements by specifying the *number* of the statement you want to delete with the Delete Watch command. (The **Y** is a mnemonic for Yank.)

You can use the asterisk (*) to represent all watch statements.

■ Examples

```
>Y 2 ;* Example 1
>

>Y * ;* Example 2
>
```

Example 1 deletes watch statement 2. Example 2 deletes all watch statements and closes the watch window.

8.6 Listing Watchpoints and Tracepoints

The Watch List command lists all previously set watchpoints and tracepoints with their assigned numbers and their current values.

This command is the only way to examine current watch statements in sequential mode. The command has little use in window mode, since watch statements are already visible in the watch window.

■ Mouse

The Watch List command cannot be executed with the mouse.

■ Keyboard

The Watch List command cannot be executed with a keyboard command.

■ Dialog

To list watch statements using a dialog command, enter a command line with the following syntax:

W

The display is the same as the display that appears in the watch window in window mode.

Note

The command letter for the Watch List command is the same as the command letter for the memory version of the Watch command when no memory size is given. The difference between the commands is that the Watch List command never takes an argument. The Watch command always requires at least one argument.

■ Example

```
>W
0) code,c   :  I
1) (float)letters/words,f   :  4.777778
2) 3F65:0B20  20 20 43 4F 55 4E 54 COUNT
3) lines==11 :  0
>
```

Chapter 9
Examining Code

9.1 Introduction 153
9.2 Set Mode Command 153
9.3 Unassemble Command 155
9.4 View Command 158
9.5 Current Location Command 161
9.6 Stack Trace Command 162

9.1 Introduction

Several CodeView commands allow you to examine program code, or data related to code. The following commands are discussed in this chapter:

Command	Action
Set Mode (**S**)	Sets format for code displays
Unassemble (**U**)	Displays assembly instructions
View (**V**)	Displays source lines
Current Location (**.**)	Displays the current location line
Stack Trace (**K**)	Displays functions or procedures

9.2 Set Mode Command

The Set Mode command sets the mode in which code is displayed. The two basic display modes are source mode, in which the program is displayed as source lines, and assembly mode, in which the program is displayed as assembly-language instructions. These two modes can be combined in mixed mode, in which the program is displayed with both source lines and assembly-language instructions.

In sequential mode, there are three display modes: source, assembly, and mixed. These modes affect the output of commands that display code (Register, Trace, Program Step, and Unassemble).

In window mode, there are two display modes: source and assembly, but there are additional options for mixing source and assembly modes and controlling the display of assembly-language instructions. The display mode affects the way the program is shown in the display window.

Source and mixed modes are only available if the executable file contains symbols in the CodeView format. Programs that do not contain symbolic information (including all **.COM** files) are displayed in assembly mode.

■ Mouse

To set the display mode with the mouse, point to View on the menu bar, press a mouse button and drag the highlight to either the Source selection for source mode, or the Assembly selection for assembly mode. Then release the button.

You can further control the display of assembly-language instructions by making selections from the Options menu. See Section 3.2.3.6, "Using the Options Menu," for more information.

■ Keyboard

To change the display mode with a keyboard command, press the F3 key. If the current mode is source, the new mode will be assembly. If the current mode is assembly, the new mode will be source. This command works in either window or sequential mode. In sequential mode, the word `source` or `assembly` is displayed to indicate the new mode.

In window mode you can further control the display of assembly-language instructions by making selections from the Options menu. See Section 3.2.3.6, "Using the Options Menu," for more information.

■ Dialog

To set the display mode from the dialog window, enter a command line with the following syntax:

S[[+ | − | &]]

If the plus sign is specified (**S+**), source mode is selected, and the word `source` is displayed.

If the minus sign is specified (**S–**), assembly mode is selected, and the word `assembly` is displayed. In window mode, the display will include any assembly options, except the Mixed Source Option, previously toggled on from the Options menu. The Mixed Source option is always turned off by the **S–** command.

If the ampersand is specified (**S&**), mixed mode is selected, and the word `mixed` is displayed. In window mode, the display will include any assembly options previously toggled on from the Options menu. In addition, the Mixed Source option will be turned on by the **S&** command.

If no argument is specified (**S**), the current mode (`source`, `assembly`, or `mixed`) is displayed.

The Unassemble command in sequential mode is an exception in that it displays mixed source and assembly with both the source (**S+**) and mixed (**S&**) modes. When you enter the dialog version of the Set Mode command, the CodeView debugger outputs the name of the new display mode: source, assembly, or mixed.

Note

80286 protected-mode mnemonics cannot be displayed in assembly or mixed mode. They will not be shown in the display window in window mode.

■ Examples

```
>S+
source
>S-
assembly
>S&
mixed
>
```

The examples show the source mode being changed to source, assembly, and mixed. In window mode, the commands change the format of the display window. In sequential mode, the commands change the output from the commands that display code (Register, Trace, Program Step, Go, Execute, and Unassemble). See the sections on individual commands for examples of how they are affected by the display mode.

9.3 Unassemble Command

The Unassemble command displays the assembly-language instructions of the program being debugged. It is most useful in sequential mode, where it is the only method of examining a sequence of assembly-language instructions. In window mode it can be used to display a specific portion of assembly-language code in the display window.

■ Mouse

The Unassemble command has no direct mouse equivalent, but you can view unassembled code at any time by changing the mode to assembly or mixed (see Section 9.2, "Set Mode Command," for more information).

■ Keyboard

The Unassemble command has no direct keyboard equivalent, but you can view unassembled code at any time by changing the mode to assembly or mixed (see Section 9.2, "Set Mode Command," for more information).

■ Dialog

To display unassembled code using a dialog command, enter a command line with the following syntax:

U [[*address* | *range*]]

The effect of the command varies depending on whether you are in sequential or window mode.

In sequential mode, if you do not specify *address* or *range*, the disassembled code begins at the current unassemble address and shows the next eight lines of instructions. The unassemble address is the address of the instruction after the last instruction displayed by the previous Unassemble command. If the Unassemble command has not been used during the session, the unassemble address is the current instruction.

If you specify an *address*, the disassembly starts at that address and shows the next eight lines of instructions. If you specify a *range*, the instructions within the range will be displayed.

The format of the display depends on the current display mode (see Section 9.2, "Set Mode Command," for more information). If the mode is source (**S+**) or mixed (**S&**), the CodeView debugger displays source lines mixed with unassembled instructions. One source line is shown for each corresponding group of assembly-language instructions. If the display mode is assembly, only assembly-language instructions are shown.

In window mode, the Unassemble command changes the mode of the display window to assembly. The display format will reflect any options previously set from the Options menu. There is no output to the dialog window. If *address* is given, the instructions in the display window will begin at

the specified address. If *range* is given, only the starting address will be used. If no argument is given, the debugger scrolls down and displays the next screen of assembly-language instructions.

Note

80286 protected-mode mnemonics cannot be displayed with the Unassemble command.

■ Examples

```
>S&      ;* Example 1
mixed
>U 0x11
4E21:0011 8BEC            MOV       BP,SP
4E21:0013 B82800          MOV       AX,0028
4E21:0016 E8060C          CALL      __chkstk (0C1F)
4E21:0019 57              PUSH      DI
4E21:001A 56              PUSH      SI
29:           char  inword = FALSE;
4E21:001B C646DC00        MOV       Byte Ptr [inword],00
31:           if (argc > 1) name = argv[1];
4E21:001F 837E0401        CMP       Word Ptr [argc],01
4E21:0023 7F03            JG        _main+18 (0028)
>S-   ;* Example 2
assembly
>U 0x11
4E21:0011 8BEC            MOV       BP,SP
4E21:0013 B82800          MOV       AX,0028
4E21:0016 E8060C          CALL      __chkstk (0C1F)
4E21:0019 57              PUSH      DI
4E21:001A 56              PUSH      SI
4E21:001B C646DC00        MOV       Byte Ptr [inword],00
4E21:001F 837E0401        CMP       Word Ptr [argc],01
4E21:0023 7F03            JG        _main+18 (0028)
>
```

Example 1 sets the mode to mixed and unassembles eight lines of disassembled code, plus whatever source lines are encountered within those lines. The display would be the same if the mode were source. Example 2 sets the mode to assembly and repeats the same command.

9.4 View Command

The View command displays the lines of a text file (usually a source module or include file). It is most useful in sequential mode, where it is the only method of examining a sequence of source lines. In window mode, the View command can be used to page through the source file or to load a new source file.

■ Mouse

To load a new source file with the mouse, point to File on the menu bar, press a mouse button and drag the highlight to the Load selection, then release the mouse. A dialog box appears, asking for the name of the file you wish to load. Type the name of the file, and press the ENTER key or a mouse button. The new file appears in the display window.

The paging capabilities of the View command have no direct mouse equivalent, but you can move about in the source file by pointing to the up or down arrows on the scroll bars and then pressing different mouse buttons. See Chapter 3, "The CodeView Display," for more information about paging with the mouse.

■ Keyboard

To load a new source file with a keyboard command, press ALT-F to open the File menu, then press ALT-L to select Load. A dialog box appears, asking for the name of the file you wish to load. Type the name of the file, and press the ENTER key. The new file appears in the display window.

The paging capabilities of the View command have no direct mouse equivalent, but you can move about in the source file by first putting the cursor in the display window with the F6 key, then pressing the PGUP, PGDN, HOME, END, and UP ARROW and DOWN ARROW keys. See Chapter 3, "The CodeView Display," for more information about paging with keyboard commands.

■ Dialog

To display source lines using a dialog command, enter a command line with the following syntax:

V [[*expression*]]

Since an address for the View command is often specified as a line number (with an optional source file), a more specific syntax for the command would be as follows:

V [[.[[*filename*:]]*linenumber*]]

The effect of the command varies depending on whether you are in sequential or window mode.

In sequential mode, the View command displays eight source lines. The starting source line is one of the following:

- The current source line if no argument is given.
- The specified *linenumber*. If *filename* is given, the specified file is loaded, and the *linenumber* refers to lines in it.
- The address that *expression* evaluates to. For example, expression could be a procedure name or an address in the *segment*:*offset* format. The code segment is assumed if no segment is given.

In sequential mode, the View command is not affected by the current display mode (source, assembly, or mixed); source lines are displayed regardless of the mode.

In window mode, if you enter the View command while the display mode is assembly, the CodeView debugger will automatically switch back to source mode. If you give *linenumber* or *expression*, the display window will be redrawn so that the source line corresponding to the given *address* will appear at the top of the source window. If you specify a *filename* with a *linenumber*, the specified file will be loaded.

If you enter the View command with no arguments, the display will scroll down one line short of a page; that is, the source line that was at the bottom of the window will be at the top.

Note

The View command with no argument is similar to pressing the PGDN key, or clicking right on the DOWN ARROW key with the mouse. The difference is that pressing the PGDN key enables you to scroll down one more line.

■ Examples

```
>V countwords    ;* Example 1
58:      char  inword;
59:      int   numread;
60:            {
61:            int   count;
62:
63:            bytes += numread;
64:            for (count = 0; count <= numread; ++count) {
65:                 char code;
>
```

Example 1 (shown in sequential mode) displays eight source lines, beginning at the function `countwords`.

```
>V .math.c:30    ;* Example 2
30:                 register int j;
31:
32:                 for (j = q; j >= 0; j--)
33:                       if (t[j] + p[j] > 9) {
34:                             p[j] += t[j] - 10;
35:                             p[j-1] += 1;
36:                       } else
37:                             p[j] += t[j];
>
```

Example 2 loads the source file `math.c` and displays eight source lines starting at line `30`.

9.5 Current Location Command

The Current Location command displays the source line or assembly-language instruction corresponding to the current program location.

■ Mouse

The Current Location command cannot be executed with the mouse.

■ Keyboard

The Current Location command cannot be executed with a keyboard command.

■ Dialog

To display the current location line using a dialog command, enter a command line with the following syntax:

.

In sequential mode, the command displays the current source line. The line is displayed regardless of whether the current debugging mode is source or assembly. If the program being debugged has no symbolic information, the command will be ignored.

In window mode, the command puts the current program location (marked with reverse video or a contrasting color) in the center of the display window. The display mode (source or assembly) will not be affected. This command is useful if you have scrolled through the source code or assembly-language instructions so that the current location line is no longer visible.

For example, if you are in window mode and have executed the program being debugged to somewhere near the start of the program, but you have scrolled the display to a point near the end, the Current Location command returns the display to the current program location.

■ Example

```
>.
for (i = 0; i <= SIZE; i++);
>
```

The example illustrates how to display the current source line in sequential mode. The same command in window mode would not produce any output, but it could change the text shown in the display window.

9.6 Stack Trace Command

The Stack Trace command allows you to display functions that have been called during program execution. The first line of the display shows the name of the current function. The succeeding lines (if any) list any other functions that were called to reach the current address.

For each function, the values of any arguments are shown in parentheses after the function name. Values are shown in the current radix (the default is decimal).

The term "stack trace" is used because, as each function is called, its address and arguments are stored (pushed) on the program stack. Therefore, tracing through the stack shows the currently active functions. For C programs, the **main** function will always be at the bottom of the stack.

The Stack Trace command also enables you to find and view the source lines where individual functions were called.

Note

This discussion refers to functions, since that is the terminology for C programs. In assembly mode, the term "procedure" may be more accurate. If you are using the CodeView debugger to debug assembly-language programs, the Stack Trace command will only work if procedures were called with the calling convention used by Microsoft languages. This calling convention is explained in Chapter 10, "Interfaces with Other Languages," of the *Microsoft C Compiler User's Guide.*

■ Mouse

To view a stack trace with the mouse, point to Calls on the menu bar and press a mouse button. The Calls menu will appear, showing the current function at the top and other functions below it in the reverse order in which they were called; for example, the first function called will be at the bottom. The values of any function arguments will be shown in parentheses following the functions.

If you want to view code at the point where one of the functions was called, hold the mouse button down and drag the highlight to the function below the one you want to view, then release the button. The cursor will move to the calling source line (in source mode) or the calling instruction (in assembly or mixed mode). In other words, the cursor will indicate the calling location in the selected function where the next-level function was called. If you select the current (top-level) function, the cursor moves to the current location in that function.

■ Keyboard

To view a stack trace with a keyboard command, press ALT-C to open the Calls menu. The menu will show the current function at the top, and other functions below it in the reverse order in which they were called; for example, the first function called will be at the bottom. The values of any function arguments will be shown in parentheses following the functions.

If you want to view code at the point where one of the functions was called, press the DOWN ARROW key to move the highlight to the function below the one you want to view, then press the ENTER key. The cursor will move to the calling source line (in assembly mode) or the calling instruction (in assembly or mixed mode). In other words, the cursor will indicate the calling location in the selected function where the next-level function was called. If you select the current (top-level) function, the cursor moves to the current location in that function.

■ Dialog

To display a stack trace with a dialog command, enter a command line with the following syntax:

K

The output from the Stack Trace dialog command lists the functions in the reverse order in which they were called. The arguments to each function

are shown in parentheses. Finally, the line number from which the function was called is shown.

You can enter the line number as an argument to the View or Unassemble command if you want to view code at the point where the function was called.

In window mode, the output from the Stack Trace dialog command appears in the dialog window. You may need the dialog version rather than the menu version, since the Calls menu can be truncated if there are too many functions or function arguments. The dialog display wraps around if necessary, so that you can see all functions and all arguments.

■ Example

```
>K
analyze(67,0), line 94
countwords(0,512), line 73
main(2,5098), line 42
>
```

In the example, the first line of output indicates that the current function is `analyze`. Its first argument currently has a decimal value of `67` and its second argument has a value of `0`. The current location in this function is `line 94`.

The second line indicates that `analyze` was called by `countwords`, and that its arguments have the values `0` and `512`. Function `analyze` was called from `line 73` of function `countwords`.

Likewise, `countwords` was called from `line 42` of `main` and its arguments have the values `2` and `5098`.

If the radix had been set to 16 or 8 using the Radix (N) command, the arguments would be shown in that radix. For example, the last line would be shown as `main(0x2,0x13ea)` in hexadecimal or `main(02,011752)` in octal.

Chapter 10
Modifying Code or Data

10.1 Introduction 167
10.2 Assemble Command 167
10.3 Enter Commands 170
10.3.1 Enter Command 174
10.3.2 Enter Bytes Command 175
10.3.3 Enter ASCII Command 175
10.3.4 Enter Integers Command 176
10.3.5 Enter Unsigned Integers Command 177
10.3.6 Enter Words Command 177
10.3.7 Enter Double Words Command 178
10.3.8 Enter Short Reals Command 179
10.3.9 Enter Long Reals Command 180
10.3.10 Enter 10-Byte Reals Command 180
10.4 Register Command 181

10.1 Introduction

The CodeView debugger provides the following commands for modifying code or data in memory:

Command	Action
Assemble (**A**)	Modifies code
Enter (**E**)	Modifies memory, usually data
Register (**R**)	Modifies registers and flags

Changes to code are temporary. You can use them for testing in the CodeView debugger, but you cannot save them or permanently change the program. To make permanent changes, you must modify the source code and recompile.

10.2 Assemble Command

The Assemble command assembles 8086-family (8086, 8087, 8088, 80186, 80287, and 80286 unprotected) instruction mnemonics and places the resulting instruction code into memory at a specified address. The only 8086-family mnemonics that cannot be assembled are 80286 protected-mode mnemonics.

■ Mouse

The Assemble command cannot be executed with the mouse.

■ Keyboard

The Assemble command cannot be executed with a keyboard command.

■ Dialog

To assemble code using a dialog command, enter a command line with the following syntax:

A[[*address*]]

If *address* is specified, the assembly starts at that address; otherwise the current assembly address is assumed.

The assembly address is normally the current address (the address pointed to by the **CS** and **IP** registers). However, when you use the Assemble command, the assembly address is set to the address immediately following the last instruction where you assembled an instruction. When you enter any command that executes code (Trace, Program Step, Go, or Execute), the assembly address is reset to the current address.

When you type the Assemble command, the assembly address is displayed. The CodeView debugger then waits for you to enter a new instruction in the standard 8086-family instruction-mnemonic form. You can enter instructions in uppercase, lowercase, or both.

To assemble a new instruction, type the desired mnemonic and press the ENTER key. The CodeView debugger assembles the instruction into memory and displays the next available address. Continue entering new instructions until you have assembled all the instructions you want. To conclude assembly and return to the CodeView prompt, press the ENTER key only.

If an instruction you enter contains a syntax error, the debugger displays the message `^ Syntax error`, redisplays the current assembly address, and waits for you to enter a correct instruction. The caret symbol in the message will point to the first character the CodeView debugger could not interpret.

The following nine rules govern entry of instruction mnemonics:

1. The far-return mnemonic is **RETF**.

2. String mnemonics must explicitly state the string size. For example, use **MOVSW** to move word strings and **MOVSB** to move byte strings.

3. The CodeView debugger automatically assembles short, near, or far jumps and calls, depending on byte displacement to the destination address. These may be overridden with the **NEAR** or **FAR** prefix, as shown in the following examples:

```
JMP       0x502
JMP       NEAR 0x505
JMP       FAR  0x50A
```

The **NEAR** prefix can be abbreviated to **NE**, but the **FAR** prefix cannot be abbreviated.

4. The CodeView debugger cannot determine whether some operands refer to a word memory location or to a byte memory location. In these cases, the data type must be explicitly stated with the prefix **WORD PTR** or **BYTE PTR**. Acceptable abbreviations are **WO** and **BY**. Examples are shown below:

```
MOV       WORD PTR [BP],1
MOV       BYTE PTR [SI-1],symbol
MOV       WO PTR [BP],1
MOV       BY PTR [SI-1],symbol
```

5. The CodeView debugger cannot determine whether an operand refers to a memory location or to an immediate operand. The debugger uses the convention that operands enclosed in square brackets refer to memory. Two examples are shown below:

```
MOV       AX,0x21
MOV       AX,[0x21]
```

The first statement moves `0x21` into `AX`. The second statement moves the data at offset `0x21` into `AX`.

6. The **DB** instruction assembles byte values directly into memory. The **DW** instruction assembles word values directly into memory, as shown in the following examples:

```
DB        1,2,3,4,"This is an example."
DW        1000,2000,3000,"Bach"
```

7. The CodeView debugger supports all forms of indirect register instructions, as shown in the following examples:

```
ADD       BX,[BP+2].[SI-1]
POP       [BP+DI]
PUSH      [SI]
```

8. All instruction-name synonyms are supported, as shown in the following examples:

```
LOOPZ     0x100
LOOPE     0x100
JA        0x200
JNBE      0x200
```

If you assemble instructions and then examine them with the Unassemble command (**U**), the CodeView debugger may show synonymous instructions, rather than the ones you assembled.

9. Do not assemble and execute 8087 or 80287 instructions if your system is not equipped with one of these math coprocessor chips. The **WAIT** instruction, for example, will cause your system to hang up if you try to execute it without the appropriate chip.

■ Example

```
>U 0x40 L 1
39B0:0040 89C3                MOV         BX,AX
>A 0x40
39B0:0040 MOV       CX,AX
39B0:0042
>U 0x40 L 1
39B0:0040 89C1                MOV         CX,AX
>
```

The example modifies the instruction at address `0x40` so that it moves data into the `CX` register instead of the `BX` register. The Unassemble command (**U**) is used to show the instruction before and after the assembly.

You can modify a portion of code for testing, as in the example, but you cannot save the modified program. You must modify your source code and recompile.

10.3 Enter Commands

The CodeView debugger has several commands for entering data to memory. You can use these commands to modify either code or data, though code can usually be modified more easily with the Assemble command (**A**). The Enter commands are listed below:

Command	Command Name
E	Enter (size is the default type)
EB	Enter Bytes
EA	Enter ASCII

EI	Enter Integers
EU	Enter Unsigned Integers
EW	Enter Words
ED	Enter Double Words
ES	Enter Short Reals
EL	Enter Long Reals
ET	Enter 10-Byte Reals

■ Mouse

The Enter commands cannot be executed with the mouse.

■ Keyboard

The Enter commands cannot be executed with keyboard commands.

■ Dialog

To enter data to memory with a dialog command, enter a command line with the following syntax:

E[[*type*]] *address* [[*list*]]

The *type* is a one-letter specifier that indicates the type of the data to be entered. The *address* indicates where the data will be entered. If no segment is given in the address, the data segment (**DS**) is assumed.

The *list* can consist of one or more expressions that evaluate to data the size of the Enter command. This data will be entered to memory at *address*. If one of the values in the list is invalid, an error message will be displayed. The values preceding the error are entered; values at and following the error are not entered.

The expressions in the list are evaluated in the current radix, regardless of the size and type of data being entered. For example, if the radix is 10 and you give the value 10 in a list with the Enter Words command, the decimal value 10 will be entered even though word values are normally entered in hexadecimal. This means that the Enter Words, Enter Integers, and Enter Unsigned Integers commands are identical when used with the list method, since 2-byte data is being entered for each command.

If *list* is not given, the CodeView debugger will prompt for values to be entered to memory. Values entered in response to prompts are accepted in hexadecimal for the Enter Bytes, Enter ASCII, Enter Words, and Enter Double Words commands. The Enter Integers command accepts signed decimal integers, while the Enter Unsigned Integers command accepts unsigned decimal integers. The Enter Short Reals, Enter Long Reals, and Enter 10-Byte Reals commands accept decimal floating-point values.

With the prompting method of data entry, the CodeView debugger prompts for a new value at *address* by displaying the address and its current value. You can then replace the value, skip to the next value, return to a previous value, or exit the command, as explained below:

- To replace the value, type the new value after the current value.
- To skip to the next value, press the SPACEBAR. Once you have skipped to the next value, you can change its value or skip to the next value. If you pass the end of the display, the CodeView debugger displays a new address to start a new display line.
- To return to the preceding value, type a backslash (\). When you return to the preceding value, the debugger starts a new display line with the address and value.
- To stop entering values and return to the CodeView prompt, press the ENTER key. You can exit the command at any time.

Sections 10.3.1–10.3.10 discuss the Enter commands in order of the size of data they accept.

■ Examples

```
>EW place 16 32 ;* Example 1
```

Example 1 shows how to enter two word-sized values at the address `place`. If the default radix (decimal) is in effect, this command enters the hexadecimal values 0x10 and 0x20.

```
>EW place                    ;* Example 2

3DA5:0B20   00F3._
```

Example 2 illustrates the prompting method of entering data. When you supply the address where you want to enter data but supply no data to be entered there, the CodeView debugger displays the current value of the address and waits for you to enter a new value. The underscore in these examples represents the CodeView cursor. You could change the value `F3`

to the new value 16 (0x10) by typing 10 (but don't press the ENTER key yet). The value must be typed in hexadecimal for the Enter Words command, as shown below:

```
>EW place                    ;* Example 2

3DA5:0B20  00F3.10_
```

You could then skip to the next value by pressing the SPACEBAR. The CodeView debugger responds by displaying the value of the next value, as shown below:

```
>EW place                    ;* Example 2

3DA5:0B20  00F3.10  4F20._
```

You could then type another hexadecimal value, such as 30:

```
>EW place                    ;* Example 2

3DA5:0B20  00F3.10  4F20.30_
```

Press the SPACEBAR to move to the next value:

```
>EW place                    ;* Example 2

3DA5:0B20  00F3.10  4F20.30  3DC1._
```

Assume you realize that the last value entered, 30, is incorrect. You really wanted to enter 20. You could return to the previous value by typing a backslash. The CodeView debugger starts a new line, starting with the previous value. Note that the backslash is not echoed:

```
>EW place                    ;* Example 2

3DA5:0B20  00F3.10  4F20.30  3DC1.
3DA5:0B22  0030._
```

Type the correct value, 20:

```
>EW place                    ;* Example 2

3DA5:0B20  00F3.10  4F20.30  3DC1.
3DA5:0B22  0030.20_
```

If this is the last value you want to enter, press the ENTER key to stop. The CodeView prompt reappears, as shown below:

```
>EW place                    ;* Example 2

3DA5:0B20   00F3.10   4F20.30   3DC1.
3DA5:0B22   0030.20
>_
```

10.3.1 Enter Command

■ Syntax

E *address* [[*list*]]

The Enter command enters one or more values into memory at the specified *address*. The data is entered in the format of the default type, which is the last type specified with a Dump, Enter, Watch Memory, or Tracepoint Memory command. If none of these commands has been entered during the session, the default type is bytes.

Use this command with caution when entering values in the list format; values will be truncated if you enter a word-sized value when the default type is actually byte. If you are not sure of the current default type, specify the size in the command.

Important

The Execute command and the Enter command have the same command letter (**E**). The difference is that the Execute command never takes an argument; the Enter command always requires at least one argument.

10.3.2 Enter Bytes Command

■ Syntax

EB *address* [[*list*]]

The Enter Bytes command enters one or more byte values into memory at *address*. The optional *list* can be entered as a list of expressions separated by spaces. The expressions are evaluated and entered in the current radix. If *list* is not given, the CodeView debugger prompts for new values, which must be entered in hexadecimal.

The Enter Bytes command can also be used to enter strings, as described in Section 10.3.3, "Enter ASCII Command."

■ Examples

```
>EB 0x100 10 20 30    ;* Example 1
>
```

If the current radix is 10, Example 1 replaces the three bytes at DS:0x100, DS:0x101, and DS:0x102 with the decimal values 10, 20, and 30.

```
>EB 0x100                    ;* Example 2

3DA5:0100  130F.A
>
```

Example 2 replaces the byte at DS:0x100 with 10 (0xA).

10.3.3 Enter ASCII Command

■ Syntax

EA *address* [[*list*]]

The Enter ASCII command works in the same way as the Enter Bytes command (**EB**) described in Section 10.3.2, "Enter Bytes Command." The *list* version of this command can be used to enter a string expression. You can include escape sequences in strings.

■ Example

```
>EA message "Can\'t open file"
>
```

In the example above, the string `Can\'t open file` is entered starting at the symbolic address `message`.

You can also use the Enter Bytes command to enter a string expression, or you can enter nonstring values using the Enter ASCII command, as described in Section 10.3.2, "Enter Bytes Command."

10.3.4 Enter Integers Command

■ Syntax

EI *address* [[*list*]]

The Enter Integers command enters one or more word values into memory at *address* using the signed-integers format. With the CodeView debugger, a signed integer can be any decimal integer between –32768 and 32767.

The optional *list* can be entered as a list of expressions separated by spaces. The expressions are entered and evaluated in the current radix. If *list* is not given, the CodeView debugger prompts for new values, which must be entered in decimal.

■ Examples

```
>EI 0x100 -10 10 -20          ;* Example 1
>
```

If the current radix is 10, Example 1 replaces the three integers at DS:0x100, DS:0x102, and DS:0x104 with the decimal values −10, 10, and −20.

```
>EI 0x100                     ;* Example 2

3DA5:0100  130F.-10
>
```

Example 2 replaces the integer at DS:0x100 with −10.

10.3.5 Enter Unsigned Integers Command

■ Syntax

EU *address* [[*list*]]

The Enter Unsigned Integers command enters one or more word values into memory at *address* using the unsigned-integers format. With the CodeView debugger, a signed integer can be any decimal integer between 0 and 65535.

The optional *list* can be entered as a list of expressions separated by spaces. The expressions are entered and evaluated in the current radix. If *list* is not given, the CodeView debugger prompts for new values, which must be entered in decimal.

■ Examples

```
>EU 0x100 10 20 30    ;* Example 1
>
```

If the current radix is 10, Example 1 replaces the three unsigned integers at DS:0x100, DS:0x102, and DS:0x104 with the decimal values 10, 20, and 30.

```
>EU 0x100                    ;* Example 2

3DA5:0100  130F.10
>
```

Example 2 replaces the integer at DS:0x100 with 10.

10.3.6 Enter Words Command

■ Syntax

EW *address* [[*list*]]

The Enter Words command enters one or more word values into memory at *address.*

The optional *list* can be entered as a list of expressions separated by spaces. The expressions are entered and evaluated in the current radix. If *list* is not given, the CodeView debugger prompts for new values, which must be entered in hexadecimal.

■ Examples

```
>EW 0x100 10 20 30    ;* Example 1
>
```

If the current radix is 10, Example 1 replaces the three words at DS:0x100, DS:0x102, and DS:0x104 with the hexadecimal values A, 14, and 1E.

```
>EW 0x100                    ;* Example 2

3DA5:0100  130F.A
>
```

Example 2 replaces the integer at DS:0x100 with 10 (0xA).

10.3.7 Enter Double Words Command

■ Syntax

ED *address* [[*list*]]

The Enter Double Words command enters one or more double-word values into memory at *address*. Double words are displayed and entered in the *segment*:*offset* address format; that is, two words separated by a colon (:). If the colon is omitted and only one word entered, only the offset portion of the address will be changed.

The optional *list* can be entered as a list of expressions separated by spaces. The expressions are entered and evaluated in the current radix. If *list* is not given, the CodeView debugger prompts for new values, which must be entered in hexadecimal.

■ Examples

```
>ED 0x100 8700:12008 ;* Example 1
>
```

If the current radix is 10, Example 1 replaces the double words at DS:0x100 with the hexadecimal address 21FC:2EE8 (8700:12008).

```
>ED 0x100                    ;* Example 2

3DA5:00C8  21FC:2EE8.2EE9
>
```

Example 2 replaces the offset portion of the double word at DS:0x100 with 0x2EE9. Since the segment portion of the address is not provided, the existing segment (0x21FC) is unchanged.

10.3.8 Enter Short Reals Command

■ Syntax

ES *address* [[*list*]]

The Enter Short Reals command enters one or more short-real values into memory at *address*.

The optional *list* can be entered as a list of real numbers separated by spaces. The numbers must be entered in decimal, regardless of the current radix. If *list* is not given, the CodeView debugger prompts for new values, which must be entered in decimal. Short-real numbers can be entered either in floating-point format or in scientific-notation format.

■ Examples

```
>ES 0x100 23.479 1/4 -1.65E+4 235    ;* Example 1
>
```

Example 1 replaces the four numbers at DS:0x100, DS:0x104, DS:0x108, and DS:0x10C with the real numbers 23.479, 0.25, -1650.0, and 235.0.

```
>ES pi                                          ;* Example 2
3DA5:0064  42 79 74 65   7.215589E+022  3.141593
>
```

Example 2 replaces the number at the symbolic address `pi` with `3.141593`.

10.3.9 Enter Long Reals Command

EL *address* [[*list*]]

The Enter Long Reals command enters one or more long-real values into memory at *address*.

The optional *list* can be entered as a list of real numbers separated by spaces. The numbers must be entered in decimal, regardless of the current radix. If *list* is not given, the CodeView debugger prompts for new values, which must be entered in decimal. Long-real numbers can be entered either in floating-point format or in scientific-notation format.

■ Examples

```
>EL 0x100 23.479 1/4 -1.65E+4 235     ;* Example 1
>
```

Example 1 replaces the four numbers at DS:0x100, DS:0x108, DS:0x110, and DS:0x118 with the real numbers `23.479`, `0.25`, `-1650.0`, and `235.0`.

```
>EL pi                                          ;* Example 2
3DA5:0064  42 79 74 65 DC 0F 49 40   5.012391E+001  3.141593
>
```

Example 2 replaces the number at the symbolic address `pi` with `3.141593`.

10.3.10 Enter 10-Byte Reals Command

■ Syntax

EL *address* [[*list*]]

The Enter 10-Byte Reals command enters one or more 10-byte-real values into memory at *address*.

The optional *list* can be entered as a list of real numbers separated by spaces. The numbers must be entered in decimal, regardless of the current radix. If *list* is not given, the CodeView debugger prompts for new values, which must be entered in decimal. The numbers can be entered either in floating-point format or in scientific-notation format.

■ Examples

```
>ET 0x100 23.479 1/4 -1.65E+4 235     ;* Example 1
>
```

Example 1 replaces the four numbers at DS:0x100, DS:0x10A, DS:0x114, and DS:0x118 with the real numbers `23.479`, `0.25`, `-1650.0`, and `235.0`.

```
>ET pi                                                ;* Example 2
3DA5:0064  42 79 74 65 DC 0F 49 40 7F BD  -3.292601E-193  3.141593
>
```

Example 2 replaces the number at the symbolic address `pi` with `3.141593`.

10.4 Register Command

The Register command has two functions. It displays the contents of the central processing unit (CPU) registers, and it can also change the values of those registers. The modification features of the command are explained in this section. The display features of the Register command are explained in Chapter 6, "Examining Data and Expressions."

■ Mouse

The only register that can be changed with the mouse is the flags register. The register's individual bits (called flags) can be set or cleared. To change a flag, first make sure the register window is open. The window can be opened by selecting Registers from the Options menu, or by pressing the F2 key.

The flag values are shown as mnemonics in the bottom of the window. Point to the flag you want to change and click either button. The mnemonic word representing the flag value will change. The mnemonics for each flag are shown in the third and fifth columns of Table 10.1. The color

or highlighting of the flag will also be reversed when you change a flag. Set flags are shown in red on color monitors and in high-intensity text on two-color monitors. Cleared flags are shown in light blue or normal text.

■ Keyboard

The registers cannot be changed with keyboard commands.

■ Dialog

To change the value of a register with a dialog command, enter a command line with the following syntax:

R [[*registername*[[=]*expression*]]]

To modify the value in a register, type the command letter **R** followed by *registername*. The CodeView debugger displays the current value of the register and prompts for a new value. Press the ENTER key if you only want to examine the value. If you want to change it, type an expression for the new value and press the ENTER key.

As an alternative, you can type both *registername* and *expression* in the same command. You can use the equal sign (=) between *registername* and *expression*, but a space has the same effect.

The register name can be any of the following names: **AX**, **BX**, **CX**, **DX**, **CS**, **DS**, **SS**, **ES**, **SP**, **BP**, **SI**, **DI**, **IP**, or **F** (for flags).

To change a flag value, supply the register name **F** when you enter the Register command. The command displays the current value of each flag as a two-letter name. The flag values are shown in Table 10.1.

Table 10.1

Flag-Value Mnemonics

Flag Name	Set Dialog	Set Window	Clear Dialog	Clear Window
Overflow	OV	overflow	NV	novrflow
Direction	DN	down	UP	up
Interrupt	EI	enable	DI	disable
Sign	NG	negative	PL	positive

Table 10.1 *(continued)*

Flag Name	Set Dialog	Set Window	Clear Dialog	Clear Window
Zero	ZR	zero	NZ	not zero
Auxiliary carry	AC	auxcarry	NA	no auxcy
Parity	PE	even	PO	odd
Carry	CY	carry	NC	no carry

At the end of the list of values, the command displays a dash (–). Enter new values after the dash for the flags you wish to change, then press the ENTER key. You can enter flag values in any order. Flags for which new values are not entered remain unchanged. If you do not want to change any flags, simply press the ENTER key.

If you enter an illegal flag name, an error message will be displayed. The flags preceding the error are changed; flags at and following the error are not changed.

■ Examples

```
>R IP 0x100 ;* Example 1
>
```

Example 1 changes the IP register to the value 256 (0x100).

```
>R AX              ;* Example 2
AX 0E00
:_
```

Example 2 displays the current value of the AX register and prompts for a new value (the underscore represents the CodeView cursor). You can now type any 16-bit value after the colon. For example, if the current radix is 10, you can enter 256 to change the AX value to 256 (0x100):

```
>R AX
AX 0E00
:256
>_
```

Examples 3 and 4 below show the command line and prompting methods of changing flag values:

```
>R F UP EI PL          ;* Example 3
>R F                   ;* Example 4
NV(OV) UP(DN) EI(DI) PL(NG) NZ(ZR) AC(NA) PE(PO) NC(CY) -OV DI ZR
>R F
OV(NV) UP(DN) DI(EI) PL(NG) ZR(NZ) AC(NA) PE(PO) NC(CY) -
>
```

With the prompting method (Example 4), the first mnemonic for each flag is the current value, and the second mnemonic (in parentheses) is the alternate value. You can enter one or more mnemonics at the dash prompt. In the example, the command is given a second time to show the results of the first command.

Chapter 11

Using System-Control Commands

11.1 Introduction 187
11.2 Help Command 187
11.3 Quit Command 188
11.4 Radix Command 189
11.5 Redraw Command 191
11.6 Screen Exchange Command 191
11.7 Search Command 192
11.8 Shell Escape Command 195
11.9 Tab Set Command 198
11.10 Redirection Commands 199
11.10.1 Redirecting CodeView Input 199
11.10.2 Redirecting CodeView Output 200
11.10.3 Redirecting CodeView Input and Output 201
11.10.4 Commands Used with Redirection 202
11.10.4.1 Comment Command 202
11.10.4.2 Delay Command 204
11.10.4.3 Pause Command 204

11.1 Introduction

This chapter discusses commands that control the operation of the CodeView debugger. The commands in this category are listed below:

Command	Action
Help (**H**)	Displays help
Quit (**Q**)	Returns to MS-DOS
Radix (**N**)	Changes radix
Redraw (@)	Redraws screen
Screen Exchange (\)	Switches to output screen
Search (/)	Searches for regular expression
Shell Escape (!)	Starts new MS-DOS shell
Tab Set (#)	Sets tab size
Redirection and related commands	Control redirection of CodeView output or input

The system-control commands are discussed in the following sections.

11.2 Help Command

The CodeView debugger has two help commands: one command accesses a complete on-line-help system. This command can only be used in window mode. The other command provides a syntax summary of dialog commands only. It can be used in either window or sequential mode.

■ Mouse

To enter the complete on-line-help system with the mouse, point to View on the menu bar, press a mouse button and drag the highlight down to the Help selection, then release the button. The initial help screen appears. See Chapter 3, "The CodeView Display," for details on using the on-line-help system. The syntax-summary screen cannot be reached with the mouse.

■ Keyboard

To enter the complete on-line-help system with a keyboard command, press the F1 key. If you are in window mode, the initial help screen appears. See Chapter 3, "The CodeView Display," for details on using the on-line-help system. If you are in sequential mode, the syntax-summary screen appears.

■ Dialog

The on-line-help system cannot be reached with a dialog command. To view the syntax summary, enter a command line with the following syntax:

```
H
```

The screen displays all CodeView dialog commands with the syntax for each. This screen is the only help available in sequential mode. It may also be useful as a quick summary in window mode.

11.3 Quit Command

The Quit command terminates the CodeView debugger and returns control to MS-DOS.

■ Mouse

To quit the CodeView debugger with the mouse, point to File on the menu, press a mouse button and drag the highlight down to the Quit selection, then release the button. The CodeView screen will be replaced by the MS-DOS screen, with the cursor at the MS-DOS prompt.

■ Keyboard

To quit the CodeView debugger with a keyboard command, press ALT-F to open the File menu, then press ALT-Q to select Quit. The CodeView screen will be replaced by the MS-DOS screen, with the cursor at the MS-DOS prompt.

■ Dialog

To quit the CodeView debugger using a dialog command, enter a command line with the following syntax:

Q

When the command is entered, the CodeView screen will be replaced by the MS-DOS screen, with the cursor at the MS-DOS prompt.

11.4 Radix Command

The Radix command changes the current radix for entering arguments and displaying the value of expressions. The default radix when you start the CodeView debugger is 10 (decimal). Radixes 8 (octal) and 16 (hexadecimal) can also be set. Binary and other radixes are not allowed.

The following seven conditions are exceptions; they are not affected by the Radix command:

1. The radix for entering a new radix is always decimal.
2. Type specifiers given with the Display Expression command or any of the Watch Statement commands override the current radix.
3. Addresses output by the Assemble, Dump, Enter, Examine Symbol, and Unassemble commands are always shown in hexadecimal.
4. In assembly mode, all values are shown in hexadecimal.
5. The display radix for Dump, Watch Memory, and Tracepoint Memory commands is always hexadecimal if the size is bytes, words, or double words, and always decimal if the size is integers, unsigned integers, short reals, long reals, or 10-byte reals.
6. The input radix for the Enter commands with the prompting method is always hexadecimal if the size is bytes, words, or double words, and always decimal if the size is integers, unsigned integers, short reals, long reals, or 10-byte reals. The current radix is used for all values given as part of a list, except real numbers, which must be entered in decimal.
7. The register display is always in hexadecimal.

■ Mouse

You cannot change the input radix with the mouse.

■ Keyboard

You cannot change the input radix using a keyboard command.

■ Dialog

To change the input radix using a dialog command, enter a command line with the following syntax:

N[[*radixnumber*]]

The *radixnumber* can be 8 (octal), 10 (decimal), or 16 (hexadecimal). The default radix when you start the CodeView debugger is 10 (decimal). If you give the Radix command with no argument, the debugger displays the current radix.

■ Examples

```
>N8                    ;* Example 1
>? prime
0153
>N10
>? prime
107
>N16
>? prime
0x6b
>

>N8                    ;* Example 2
>? 34,i
28
>N10
>? 28,i
28
>N16
>? 1C,i
28
>
```

In Example 1, the value of a variable is displayed in each radix. In Example 2, the same number is entered in different radixes, but the `i` type specifier is used to display the result as a decimal integer in all three cases. See Chapter 6, "Examining Data and Expressions," for more information on type specifiers.

11.5 Redraw Command

The Redraw command can only be used in window mode. It redraws the CodeView screen. This command is seldom necessary, but you might need it if the output of the program being debugged temporarily disturbs the CodeView display.

■ Mouse

You cannot redraw the screen using the mouse.

■ Keyboard

You cannot redraw the screen using a keyboard command.

■ Dialog

To redraw the screen using a dialog command, enter a command line with the following syntax:

```
@
```

11.6 Screen Exchange Command

The Screen Exchange command allows you to temporarily switch from the debugging screen to the output screen.

The CodeView debugger will use either screen flipping or screen swapping to store the output and debugging screens. See Chapter 2, "Getting Started," for an explanation of flipping and swapping.

■ Mouse

To switch to the output screen with the mouse, point to View on the menu bar, press a mouse button and drag the highlight down to the Output selection, then release the button. The output screen appears. Press any key or a mouse button when you are ready to return to the debugging screen.

■ Keyboard

To switch to the output screen with a keyboard command, press the F4 key. The output screen appears. Press any key when you are ready to return to the debugging screen. This command works in either window or sequential mode.

■ Dialog

To execute the Execute command from the dialog window, enter a command line with the following syntax:

\

The output screen appears. Press any key when you are ready to return to the debugging screen.

11.7 Search Command

The Search command allows you to search for a regular expression in a source file. The expression to be found is specified either as an argument to a dialog command or in a dialog box. Once you have found an expression, you can also search for the next or previous occurrence of the expression.

Regular expressions are a method of specifying variable text patterns. A pattern can be used to search for text strings that match the pattern. This method comes from the XENIX and UNIX operating systems, and is similar to the MS-DOS method of using wild-card characters in file names. Regular expressions are explained in detail in Appendix B.

You can use the Search command without understanding regular expressions. Since text strings are the simplest form of regular expressions, you

can simply enter a string of characters as the expression you want to find. For example, you could enter `count` if you wanted to search for the word "count" in the source file.

The following characters have special meanings in regular expressions: backslash (\), asterisk (*), left bracket ([), period (.), dollar sign ($), and caret (^). To find strings containing these characters, you must precede the characters with a backslash; this cancels their special meanings.

For example, you would use `\\n` to find `\n` or use `buffer\[count]` to find `buffer[count]`. The period in some member-selection expressions and the caret in the XOR operator and the XOR assignment operator must also be preceded by a backslash.

Important

When you search for the next occurrence of a regular expression, the CodeView debugger searches to the end of the file, then wraps around and begins again at the start of the file. This can have unexpected results if the expression occurs only once. When you give the command repeatedly, nothing seems to happen. Actually, the debugger is repeatedly wrapping around and finding the same expression each time.

■ Mouse

To find a regular expression with the mouse, point to Search on the menu bar, press a mouse button and drag the highlight down to the Find selection, then release the button. A dialog box appears, asking for the regular expression to be found. Type the expression, and press either the ENTER key or a mouse button. The CodeView debugger starts searching at the current cursor position and puts the cursor at the next line containing the regular expression. An error message appears if the expression is not found. If you are in assembly mode, the debugger automatically switches to source mode when the expression is found.

After you have found a regular expression, you can search for the next or previous occurrence of the expression. Point to Search on the menu bar, press a mouse button and drag the highlight down to the Next or Previous selection, then release the button. The cursor will move to the next or previous match of the expression.

You can also search the executable code for a label (such as a function name or an assembly-language label). Point to Search on the menu bar, press a mouse button and drag the highlight down to the Label selection, then release the button. A dialog box appears, asking for the label to be found. Type the label name and press either the ENTER key or a mouse button. The cursor will move to the line containing the label. This selection differs from other search selections because it searches executable code rather than source code. The CodeView debugger will switch to assembly mode, if necessary, to display a label in a library routine or assembly-language module.

■ Keyboard

To find a regular expression with a keyboard command, press ALT-S to open the Search menu, then press ALT-F to select Find. A dialog box appears, asking for the regular expression to be found. Type the expression, and press the ENTER key. The CodeView debugger starts searching at the current cursor position and puts the cursor at the next line containing the regular expression. An error message appears if the expression is not found. If you are in assembly mode, the debugger automatically switches to source mode when the expression is found.

After you have found a regular expression, you can search for the next or previous occurrence of the expression. Press ALT-S to open the Search menu, then press ALT-N to select Next or ALT-P to select Previous. The cursor will move to the next or previous match of the expression.

You can also search the executable code for a label (such as a function name or an assembly-language label). Press ALT-S to open the Search menu, then press ALT-L to select Label. A dialog box appears, asking for the label to be found. Type the label name and press the ENTER key. The cursor will move to the line containing the label. This selection differs from other search selections because it searches executable code rather than source code. The CodeView debugger will switch to assembly mode, if necessary, to display a label in a library routine or assembly-language module.

■ Dialog

To find a regular expression using a dialog command, enter a command line with the following syntax:

/[[*regularexpression*]]

If *regularexpression* is given, the CodeView debugger searches the source file for the first line containing the expression. If no argument is given, the debugger searches for the next occurrence of the last regular expression specified.

In window mode, the CodeView debugger starts searching at the current cursor position and puts the cursor at the next line containing the regular expression. In sequential mode, the debugger starts searching at the last source line displayed. It puts the source line where the expression is found on the screen. An error message appears if the expression is not found. If you are in assembly mode, the CodeView debugger automatically switches to source mode when the expression is found.

You cannot search for a label with the dialog version of the Search command, but using the View command with the label as an argument has the same effect.

11.8 Shell Escape Command

The Shell Escape command allows you to exit the CodeView debugger to an MS-DOS shell. You can execute MS-DOS commands or programs from within the debugger, or you can exit from the debugger to MS-DOS while retaining your current debugging context.

The Shell Excape command works by saving the current processes in memory and then executing a second copy of **COMMAND.COM**. The **COMSPEC** environment variable is used to locate a copy of **COMMAND.COM**.

Opening a shell requires a significant amount of free memory (usually more than 200K). This is because the CodeView debugger, the symbol table, **COMMAND.COM**, and the program being debugged must all be saved in memory. If you do not have enough memory, an error message will appear. Even if you have enough memory to start a new shell, you may not have enough memory left to execute large programs from the shell.

If you change directories while working in the shell, make sure you return to the original directory before returning to the CodeView debugger. If you don't, the debugger may not be able to find and load source files when it needs them.

Note

In order to use the Shell Escape command, the executable file being debugged must release the memory it does not need. Programs compiled with the Microsoft C Compiler do this automatically if the C start-up code has been executed. You must execute into the program before using the Shell Escape command; for example, enter `G main` after starting the CodeView debugger.

You cannot use the Shell Escape command with assembler programs unless the program specifically releases memory using the MS-DOS function call 0x4A (Set Block). The same thing can be accomplished by linking the assembler program with the **/CPARMAXALLOC** link option. If the program has not released memory, the CodeView debugger will print this message: `Not enough memory`.

■ Mouse

To open an MS-DOS shell with the mouse, point to File on the menu bar, press a mouse button and drag the highlight down to the Shell selection, then release the button. If there is enough memory to open the shell, the MS-DOS screen will appear. You can execute any MS-DOS internal command or any program. When you are ready to return to the debugging session, type the command `exit` (in any combination of uppercase and lowercase letters). The debugging screen will appear with the same status it had when you left it.

■ Keyboard

To open an MS-DOS shell using a keyboard command, press ALT-F to open the File menu, then press ALT-S to select Shell. If there is enough memory to open the shell, the MS-DOS screen will appear. You can execute any MS-DOS internal command or any program. When you are ready to return to the debugging session, type the command `exit` (in any combination of uppercase and lowercase letters). The debugging screen will appear with the same status it had when you left it.

■ Dialog

To open an MS-DOS shell using a dialog command, enter a command line with the following syntax:

! [[*command*]]

If you want to exit to MS-DOS and execute several programs or commands, enter the command with no arguments. The CodeView debugger executes a new copy of **COMMAND.COM** and the MS-DOS screen appears. You can run programs or MS-DOS internal commands. When you are ready to return to the debugger, type the command `exit` (in any combination of uppercase and lowercase letters). The debugging screen will appear with the same status it had when you left it.

If you want to execute a program or MS-DOS internal command from within the CodeView debugger, enter the Shell Escape command (!) followed by the name of the command or program you want to execute. The output screen appears and the debugger executes the command or program. When the output from the command or program is finished, the message `Press any key to continue...` appears at the bottom of the screen. Press a key to make the debugging screen reappear with the same status it had when you left it.

■ Examples

```
>!                  ;* Example 1

>!DIR a:*.c         ;* Example 2

>!CHKDSK a:         ;* Example 3
```

In Example 1, the CodeView debugger saves the current debugging context and executes a copy of **COMMAND.COM**. The MS-DOS screen appears and you can enter any number of commands. To return to the debugger, enter `exit`.

In Example 2, the MS-DOS internal command `DIR` is executed with the argument `a:*.c`. The directory listing will be followed by a prompt telling you to press any key to return to the CodeView debugging screen.

In Example 3, the MS-DOS external command `CHKDSK` is executed, and the status of the disk in Drive A is displayed in the dialog window. The program name specified could be for any executable file, not just for an MS-DOS external program.

11.9 Tab Set Command

The Tab Set command sets the width in spaces that the CodeView debugger fills for each tab character. The default tab is eight spaces. You might want to set a smaller tab size if your source code has so many levels of indentation that source lines extend beyond the edge of the screen. This command has no effect if your source code was written with an editor that indents with spaces rather than with tab characters.

■ Mouse

You cannot set the tab size using the mouse.

■ Keyboard

You cannot set the tab size using a keyboard command.

■ Dialog

To set the tab size using a dialog command, enter a command line with the following syntax:

number

The *number* is the new number of characters for each tab character. In window mode, the screen will be redrawn with the new tab width when you enter the command. In sequential mode, any output of source lines will reflect the new tab size.

■ Example

```
>.
32:                    for (j = q; j >= 0; j--)
>#4
>.
32:            for (j = q; j >= 0; j--)
>
```

In this example, the Source Line (.) command is used to show the source line with the default tab width of eight spaces. Next the Tab Set command is used to set the tab width to four spaces. The Source Line command then shows the same line.

11.10 Redirection Commands

The CodeView debugger provides several options for redirecting commands from or to devices or files. In addition to the redirection commands, several other commands are only relevant when used with redirected files. The redirection commands and related commands are discussed in sections 11.10.1–11.10.4.3.

- **Mouse**

None of the redirection or related commands can be executed with the mouse.

- **Keyboard**

None of the redirection or related commands can be executed with keyboard commands.

- **Dialog**

The redirection commands are entered with dialog commands, as shown in sections 11.10.1–11.10.4.3.

11.10.1 Redirecting CodeView Input

- **Syntax**

< *devicename*

The Redirected Input command causes the CodeView debugger to read all subsequent command input from a device, such as another terminal or a file. The sample session provided with the debugger is an example of commands being redirected from a file.

■ Examples

```
><COM1              ;* Example 1

><INFILE.TXT        ;* Example 2
```

Example 1 redirects commands from the device (probably a remote terminal) designated as `COM1` to the CodeView terminal.

Example 2 redirects command input from file `INFILE.TXT` to the CodeView debugger. You might use this command to prepare a CodeView session for someone else to run. You create a text file containing a series of CodeView commands separated by carriage-return–line-feed combinations or semicolons. When you redirect the file, the debugger will execute the commands to the end of the file. If you want the user to be able to continue editing after the session, the last command in the file should be `< CON` so that command input will be returned to the CodeView console or terminal. One way to create such a file is to redirect commands from the CodeView debugger to a file (see Section 11.10.3) and then edit the file to eliminate the output and to add comments.

11.10.2 Redirecting CodeView Output

■ Syntax

[[**T**]]**>**[[**>**]] *devicename*

The Redirected Output command causes the CodeView debugger to write all subsequent command output to a device, such as another terminal, a printer, or a file. The term "output" includes not only the output from commands, but the command characters that are echoed as you type them.

The optional **T** indicates that the output should be echoed to the CodeView screen. If you do not use the **T**, you will not be able to see your commands as you type them. Normally, you will want to use the **T** if you are redirecting output to a file, so that you can see what you are typing. However, if you are redirecting output to another terminal, you may not want to see the output on the CodeView terminal.

The optional second greater-than symbol appends the output to an existing file. If you redirect output to an existing file without this symbol, the existing file will be replaced.

■ Examples

```
>>COM1              ;* Example 1

>T>OUTFILE.TXT     ;* Example 2
.
.
.
>>CON
.
.
.
>T>>OUTFILE.TXT ;* Example 3
```

Example 1 output is redirected to the device designated as COM1 (probably a remote terminal). One situation in which you might want to do this is when you are debugging a graphics program and want CodeView commands to be displayed on a remote terminal at the same time that the program display appears on the originating terminal.

In Example 2, output is redirected to the file OUTFILE.TXT. You might want to do this in order to keep a permanent record of a CodeView session. Note that the optional **T** is used so that the session will be echoed to the CodeView screen as well as to the file. After redirecting some commands to a file, output is returned to the console (terminal) with the command >CON. If, later in the session, you want to redirect more commands to the same file, use the double greater-than symbol, as in Example 3, to append the output to the existing file.

11.10.3 Redirecting CodeView Input and Output

■ Syntax

= *devicename*

The Redirected Input and Output command causes the CodeView debugger to write all subsequent command output to a device and to simultaneously receive input from the same device. This is only practical if the device is a remote terminal.

Redirecting input and output works best if you start in sequential mode (using the /**T** option), since this eliminates unnecessary screen exchanges. The CodeView debugger's window interface has little purpose in this situation, since the remote terminal can only act as a sequential (nonwindow) device.

■ Example

```
>=COM1
```

In this example, output and input are redirected to the device designated as COM1. This would be useful if you wanted to enter debugging commands and see the debugger output on a remote terminal, while entering program commands and viewing program output on the terminal where the debugger is running.

11.10.4 Commands Used with Redirection

The following commands are intended for use when redirecting commands to or from a file. Although they are always available, these commands have little practical use during a normal debugging session.

Command	Action
Comment (*)	Displays comment
Delay (:)	Delays execution of commands from a redirected file
Pause (")	Interrupts execution of commands from a redirected file until a key is pressed

11.10.4.1 Comment Command

■ Syntax

**comment*

The Comment command is an asterisk (*) followed by text. The CodeView debugger echoes the text of the comment to the screen (or other output device). This command is useful in combination with the redirection commands when saving a commented session, or when writing a commented session that will be redirected to the debugger.

■ Examples

```
>T>OUTPUT.TXT          ;* Example 1
>* Dump first 20 bytes of screen buffer
>D OxB800:0 L 20
B800:0000  54 17 6F 17 20 17 72 17 65 17 74 17 75 17 72 17  T.o. .r.e.t.u.r.
B800:0010  6E 17 20 17                                      n. .
>
```

In Example 1, the user is sending a copy of a CodeView session to file OUTPUT.TXT. Comments are added to explain the purpose of the command. The text file will contain commands, comments, and command output.

Example 2 below illustrates another way to use the Comment command. You can put comments into a text file of commands that will be executed automatically when you redirect the file into the CodeView debugger. For example, you might use an editing program to create the following text file called INPUT.TXT:

```
* Dump first 20 bytes of screen buffer
D OxB800:0 L 20
.
.
.
< CON
```

When you read the file into the debugger, using the Redirected Input command, you will see the comment, then the output from the command. The output is shown in Example 2 below:

```
><INPUT.TXT            ;* Example 2
>* Dump first 20 bytes of screen buffer
>D OxB800:0 L 20
B800:0000  54 17 6F 17 20 17 72 17 65 17 74 17 75 17 72 17  T.o. .r.e.t.u.r.
B800:0010  6E 17 20 17                                      n. .
.
.
.
>< CON
>
```

11.10.4.2 Delay Command

■ Syntax

```
:
```

The Delay command interrupts execution of commands from a redirected file and waits about half a second before continuing. You can put multiple Delay commands on a single line to increase the length of the delay. The delay is the same length, regardless of the processing speed of the computer.

■ Example

```
: ;* That was a short delay...
::::: ;* That was a longer delay...
```

In this example from a text file that might be redirected into the CodeView debugger, the Delay command is used to slow execution of the redirected file.

11.10.4.3 Pause Command

■ Syntax

```
"
```

The Pause command interrupts execution of commands from a redirected file and waits for the user to press a key. Execution of the redirected commands begins as soon as a key is pressed.

■ Example

```
* Press any key to continue
"
```

In this example from a text file that might be redirected into the CodeView debugger, a comment command is used to prompt the user to press a key. Then the Pause command is used to halt execution until the user responds.

The output will look like the following when the text is redirected into the debugger:

```
>* Press any key to continue
>"
```

The next CodeView prompt will not appear until the user presses a key.

Appendixes

CodeView Appendixes

A Command and Mode Summary 209

B Regular Expressions 221

C Error Messages 229

Appendix A
Command and Mode Summary

A.1 Introduction 211
A.2 Modes 211
A.3 Options 212
A.4 Window Commands 213
A.5 Dialog Commands 215
A.6 Type Specifiers 218

A.1 Introduction

This appendix summarizes the CodeView debugger's modes, options, and commands.

A.2 Modes

Many CodeView commands and options deal with switching between modes. The debugger has four types of modes: screen modes, debugging modes, display modes, and exchange modes. The modes for each of these types are shown in Table A.1.

Table A.1

CodeView Modes

Type	Mode	Purpose	Command or Option
Screen	Debugging	Debugs program	Default
	Output	Views program output	F4 Screen Exchange (\) Output from View menu
	Help	Views command and option summary	F1 Help from View menu
Debugging	Window	Views program in windows	Default for IBM **/W** option
	Sequential	Views program sequentially	Default for non-IBM **/T** option
Display	Source	Displays source lines	Default start-up for C programs F3 **S+** Source from View menu

Table A.1 *(continued)*

Type	Mode	Purpose	Command or Option
	Assembly	Displays assembly instructions	Default start-up for programs with no symbolic information F3 **S–** Assembly from View menu
	Mixed	Displays source and assembly	**S&** Mixed from Options menu
Exchange	Flipping	Flips between video pages	Default for IBM **/F** option
	Swapping	Swaps between buffers	**/S** option

A.3 Options

The following start-up options are available with the CodeView debugger:

Option	Effect
/B	Starts in black and white with CGA
/C*commands*	Executes commands on start-up
/F	Starts with screen flipping (exchanges screens by flipping video pages)
/M	Disables the mouse
/T	Starts in sequential mode
/S	Starts with screen swapping (exchanges screens by changing buffers)
/W	Starts in window mode
/43	Starts in EGA 43-row mode

A.4 Window Commands

Table A.2 shows the keyboard and mouse versions of each window command.

Table A.2

Window Commands

Action	Keyboard	Mouse
Go to help screen	F1	Help from View menu
Open register window	F2	Registers from Options menu
Toggle source/assembly	F3	Source/Assembly from View menu
Switch to output screen	F4	Output from View menu
Go	F5	Click either on Go!
Switch to display/dialog	F6	None
Execute to here	F7 at location	Click right at location
Trace through procedure	F8	Click left on Trace!
Set breakpoint here	F9 at location	Click left at location
Step over procedure	F10	Click right on Trace!
Change flag	None	Click either on flag
Move separator line up	CONTROL-U	Drag line up
Move separator line down	CONTROL-D	Drag line down
Scroll up line in window	None	Click left on up arrow
Scroll up page in window	PGUP	Click right on up arrow
Scroll to top of window	HOME	Click both on up arrow
Scroll down line in window	None	Click left on down arrow
Scroll down page in window	PGDN	Click right on down arrow
Scroll to bottom of window	END	Click both on down arrow
Move cursor	UP ARROW/ DOWN ARROW	None

Table A.3 lists and explains the selections on the CodeView menus.

Table A.3
Menu Selections

Menu	Selection	Action
File	Load...	Loads new text or source file
	Shell	Starts new DOS shell
	Quit	Quits the CodeView debugger
Search	Find...	Finds first regular expression
	Next	Finds next regular expression
	Previous	Finds previous regular expression
	Label...	Finds function or code label
View	Help	Opens on-line-help screen
	Source	Displays source lines
	Assembly	Displays assembly
	Output	Switches to output screen
	Evaluate...	Evaluates expression
Run	Start	Restarts current program and runs
	Restart	Restarts current program
	Execute	Executes in slow motion
	Clear Breakpoints	Clears all breakpoints
Watch	Add Watch...	Sets watch expression
	Watchpoint...	Sets watchpoint
	Tracepoint...	Sets tracepoint
	Delete Watch...	Deletes watch statement
Options	Flip/Swap	Toggles screen exchange
	Mix Source	Toggles mixed source and instructions
	Symbols	Toggles symbolic reference to variables
	Bytes Coded	Toggles byte display for instructions
	Registers	Toggles register window
	Case Sense	Toggles case sensitivity of symbols
Calls	*function*	Goes to *function* call line

A.5 Dialog Commands

The CodeView dialog commands and the syntax for each are listed alphabetically in this section. Many of the commands require a type to indicate the size of the data accepted by the command. The types used by the Dump, Enter, Watch Memory, and Tracepoint commands are listed below in order of their data size:

Type	Description
No type	The default type (the last one used with a Dump, Enter, Watch Memory, or Tracepoint Memory command, or byte if none of these commands has been used)
A (ASCII)	ASCII (8-bit) characters
B (Byte)	Byte (8-bit) hexadecimal values
I (Integer)	Integer (16-bit) decimal values; equivalent to C **int** (signed) on MS-DOS systems
U (Unsigned)	Unsigned (8-bit) decimal values; equivalent to C **unsigned**
W (Word)	Word (16-bit) hexadecimal values
D (Double Word)	Double-word (32-bit) hexadecimal values
S (Short Real)	Short-real (32-bit) values; equivalent to C **float**
L (Long Real)	Long-real (64-bit) values; equivalent to C **double**
T (10-Byte Real)	10-byte-real (80-bit) values

The CodeView dialog commands and the syntax for each are shown in Table A.4.

Table A.4
Dialog Commands

Name	Syntax	Description
Assemble	**A** [[*address*]]	Assembles mnemonics starting at *address*
Breakpoint Clear	**BC** [[*list*]]	Clears breakpoints in *list*
Breakpoint Disable	**BD** [[*list*]]	Disables breakpoints in *list*
Breakpoint Enable	**BE** [[*list*]]	Enables breakpoints in *list*
Breakpoint List	**BL**	Lists breakpoints with status of each
Breakpoint Set	**BP** [[*address* [[*count*]] [[" *cmds*"]]]	Sets breakpoint at *address*; *count* is pass count; *cmds* are commands to be executed at each break
Comment	*	Displays explanatory text
Delay	:	Delays execution of redirected commands (may be repeated for longer delays)
Display Expression	? *expression*[[,*format*]]	Displays *expression* in *format*
Dump	**D**[[*type*]] [[*range*]]	Dumps memory range in *type* format
Enter	**E**[[*type*]] *address* [[*list*]]	Enters memory value in *type* format
Examine Symbols	**X?**[[*mod!*]][[*func.*]][[*sym*]][[*]]	Displays specified symbols
Execute	E	Executes in slow motion
Go	**G** [[*address*]]	Executes to *address* or to end
Help	**H**	Displays dialog commands and syntax
Pause	"	Interrupts execution of redirected commands and waits for keystroke

Table A.4 *(continued)*

Name	Syntax	Description
Program Step	**P** ⟦*count*⟧	Executes source lines or instructions, stepping over function, procedure, and interrupt calls; repeats *count* times
Quit	**Q**	Exits and returns to MS-DOS
Radix	**N**⟦*radix*⟧	Sets input radix
Redirection	⟦**T**⟧>⟦>⟧*device* <*device* =*device*	Redirects input or output to or from *device*
Redraw	@	Redraws the screen
Register	**R**⟦*register* ⟦⟦=⟧*expression*⟧⟧	Displays registers and flags, or sets new registers and flags
Load	**L** ⟦*arguments*⟧	Restarts program
Screen Exchange	\	Exchanges the CodeView and output screens
Search	/⟦*regularexpression*⟧	Searches for a regular expression
Set Mode	**S**⟦+|-|&⟧	Sets display mode to source, assembly, or mixed
Shell Escape	!⟦*command*⟧	Escapes to a new MS-DOS shell
Current Location	.	Displays the current source line
Stack Trace	**K**	Displays active functions on the stack
Tab Set	# *number*	Sets *number* of spaces for each tab character
Trace	**T** ⟦*count*⟧	Executes source lines or instructions, tracing into function, procedure, or interrupt calls; repeats *count* times

Table A.4 *(continued)*

Name	Syntax	Description
Tracepoint	**TP?** *expression*[[,*format*]] **TP**[[*type*]] [[*range*]]	Breaks when *expression* or *range* changes; displays in watch window
Unassemble	**U** [[*range*]]	Displays unassembled instructions
View	**V** [[*address*]]	Displays source lines
Watch	**W?** *expression*[[,*format*]] **W**[[*type*]] [[*range*]]	Displays *expression* or *range* in watch window
Watch Delete	**Y** *number*	Deletes (yanks) watch statements
Watch List	**W**	Lists tracepoints and watchpoints
Watchpoint	**WP?** *expression*[[,*format*]]	Breaks when *expression* is true; displays in watch window
8087	**7**	Displays 8087 registers

A.6 Type Specifiers

Several commands allow you to specify the format in which expression values are displayed. The following is the syntax for commands that can have formatted output:

? *expression*[[,*format*]]	Expression command
W? *expression*[[,*format*]]	Watch command
WP? *expression*[[,*format*]]	Watchpoint command
TP? *expression*[[,*format*]]	Tracepoint command

The *format* in these commands can be a **printf** type specifier from among those listed in Table A.5.

Table A.5
Type Specifiers

Character	Argument Type	Output Format
d	Integer	Signed decimal integer
i	Integer	Signed decimal integer
u	Integer	Unsigned decimal integer
o	Integer	Unsigned octal integer
x ¦ **X**	Integer	Hexadecimal integer
f	Floating point	Signed value in floating-point decimal format with six decimal places
e ¦ **E**	Floating point	Signed value in scientific-notation format with up to six decimal places (trailing zeros or decimal point truncated)
g ¦ **G**	Floating point	Signed value with floating-point decimal or scientific notation, whichever is more compact
c	Character	Single character
s	String	Characters printed up to the first null character

The prefix **h** can be used with the integer type specifiers (**d**, **o**, **u**, **x**, and **X**) to specify a **short int**. The prefix **l** can be used with the same types to specify a **long int**.

Appendix B
Regular Expressions

B.1 Introduction 223
B.2 Special Characters in Regular Expressions 223
B.3 Searching for Special Characters 224
B.4 Using the Period 224
B.5 Using Brackets 225
B.5.1 Using the Dash within Brackets 225
B.5.2 Using the Caret within Brackets 226
B.5.3 Matching Brackets within Brackets 226
B.6 Using the Asterisk 226
B.7 Matching the Start or End of a Line 227

B.1 Introduction

Regular expressions are used to search for variable text strings. Special characters can be used within regular expressions to specify groups of characters to be searched for.

Regular expressions come from the XENIX and UNIX operating systems, where they can be used in search-and-replace commands. Since the CodeView debugger never needs to replace text, its use of regular expressions is a subset of the XENIX and UNIX regular-expression syntax.

This appendix explains all the special characters you can use to form regular expressions, but you do not need to learn the whole system to use CodeView Search commands. The simplest form of regular expression is simply a text string. For example, if you wanted to search for all instances of the symbol `count`, you could specify `count` as the string to be found.

If you only want to search for simple strings, you do not need to read this entire appendix, but you should know how to search for strings containing the special characters used in regular expressions. See Section B.3 for more information.

B.2 Special Characters in Regular Expressions

The following characters have special meanings in regular expressions:

Character	Purpose
Backslash (\)	Removes the special characteristics of the following characters: backslash (\), period (.), caret (^), dollar sign ($), asterisk (*), and left bracket ([)
Period (.)	Matches any character
Caret (^)	Matches beginning of line
Dollar sign ($)	Matches end of line
Asterisk (*)	Matches any number of repetitions of the previous character
Brackets ([ ])	Match characters specified within the brackets; the following special characters may be used inside brackets:

Caret (^)	Reverses the function of the brackets; that is, matches any character except those specified within the brackets
Dash (–)	Matches characters in ASCII order between (inclusive) the characters on either side of the minus sign

B.3 Searching for Special Characters

If you need to match one of the special characters used in regular expressions, you must precede it with a backslash when you specify a search string. The special characters are the asterisk (*), backslash (\), left bracket ([), caret (^), dollar sign ($), and period (.).

For example, the regular expression `\*name` matches `*name`. The backslash is necessary because the indirection operator (*) is a special character in regular expressions. Similarly, you could use `\\n` to search for the newline character (\n), or use `buffer\[count]` to find `buffer[count]`. Note that the backslash is only necessary for the left bracket; the right bracket is not considered a special character.

Backslashes may also be required to search for the XOR operator (^), XOR assignment operator (^=), periods in member-selection expressions, or the dollar sign ($) in variable names.

B.4 Using the Period

A period in a regular expression matches any single character. This corresponds to the question mark (?) used in specifying MS-DOS file names.

For example, you could use the regular expression `ato.` to search for any of the functions `atof`, `atoi`, or `atol`. You could use the expression `x.y` to search for strings such as `x+y`, `x-y`, or `x<y`. If your programming style is to put a space between variables and operators, you could use the regular expression `x . y` for the same purpose.

Note that when you use the period as a wild card, you will find the strings you are looking for, but you may also find other strings that you aren't interested in. You can use brackets to be more exact about the strings you want to find.

B.5 Using Brackets

You can use brackets to specify a character or characters you want to match. Any of the characters listed within the brackets is an acceptable match. This method is more exact than using a period to match any character.

For example, the regular expression `x[-+/*]y` matches `x+y`, `x-y`, `x/y`, or `x*y`, but not `x=y` or `xzy`. The regular expression `count[12]` matches `count1` and `count2`, but not `count3`. Similarly, `\\[ntvbrfa'"\0x]` matches any escape sequence.

Most regular-expression special characters have no special meaning when used within brackets. The only special characters within brackets are the dash (–), caret (^), and right bracket (]). Even these characters only have special meanings in certain contexts, as explained in sections B.5.1–B.5.3.

B.5.1 Using the Dash within Brackets

The dash can be used within brackets to specify a group of sequential ASCII characters. For example, the regular expression `[0-9]` matches any digit; it is equivalent to `[0123456789]`. Similarly, `[a-z]` matches any lowercase letter, and `[A-Z]` matches any uppercase letter.

You can combine ASCII ranges of characters with other listed characters. For example, `[A-Za-z ]` matches any uppercase or lowercase letter or a space.

The dash only has this special meaning if you use it to separate two ASCII characters. It has no special meaning if used directly after the starting bracket or directly before the ending bracket. This means that you must be careful where you place the dash (minus sign) within brackets.

For example, you might use the regular expression `[+-/*]` to match the characters +, -, /, and *. However, this does not give the intended result.

Instead it matches the characters between + and / and also the character *. To specify the intended characters, put the dash first or last in the list: [-+/*] or [+/*-].

B.5.2 Using the Caret within Brackets

If used as the first character within brackets, the caret (^) reverses the meaning of the brackets. That is, any character except the ones in brackets will be matched. For example, the regular expression [^0-9] matches any character that is not a digit. Specifying the characters to be excluded is often more concise than specifying the characters you want to match.

If the caret is not in the first position within the brackets, it is treated as an ordinary character. For example, the expression [0-9^] matches any digit or a caret.

B.5.3 Matching Brackets within Brackets

Sometimes you may want to specify the bracket characters as characters to be matched. This is no problem with the left bracket; it is treated as a normal character. However, the right bracket is interpreted as the end of the character list rather than as a character to be matched.

If you want the right bracket to be matched, you must make it the first character after the initial left bracket. For example, the regular expression []#![@%] matches either bracket character or any of the other characters listed within the brackets. However, if you changed the order of just one of the characters (to [#]![@%]), the meaning would be changed so that you would be specifying two groups of characters in brackets: [#] and [@%].

B.6 Using the Asterisk

The asterisk (*) is used following a character, to match a repeated sequence of that character. The character in the text being matched may be repeated once, zero times, or numerous times.

For example, the regular expression `for *(test` will match any of the following strings:

```
for (test
for        (test
for(test
```

This is convenient if the text you are searching might contain some spaces, but you don't know the exact number. (Be careful in this situation: you can't be sure if the text contains a series of spaces or a tab.) Note that the last example contains zero repetitions of the space character.

You might also use the asterisk to search for a symbol when you aren't sure of the spelling. For example, you could use `first*ime` if you aren't sure if the identifier you are searching for is spelled `firsttime` or `firstime`.

One particularly powerful use of the asterisk is to combine it with the period (`.*`). This combination searches for any group of characters, and is similar to the asterisk used in specifying MS-DOS file names. For example, the expression `(.*)` matches `(test)`, `(response=='Y')`, `(x=0;x<=20;x++)`, or any other string that starts with a left parenthesis and ends with a right parenthesis.

You can use brackets with the asterisk to search for a sequence of repeated characters of a given type. For example, `\[[0-9]*]` matches integer constants within brackets (`[1353]` or `[3]`), but does not match symbols within brackets (`[count]`). Empty brackets (`[]`) are also matched, since the characters in the brackets are repeated zero times.

B.7 Matching the Start or End of a Line

In regular expressions, the caret (`^`) matches the start of a line, while the dollar sign (`$`) matches the end of a line.

For example, the regular expression `^int  *` matches **int** declarations that start lines, but not indented **int** declarations. (Note that there are two spaces, so that at least one space is required for a match.) Similarly, `)$` matches a right parenthesis at the end of a line, but not a right parenthesis within a line.

You can combine both symbols to search for entire lines. For example, `^{$` matches any line consisting of only a left curly bracket in the left margin, and `^$` matches blank lines.

Error Messages

Appendix C
Error Messages

The CodeView debugger displays an error message whenever it detects a command it cannot execute. Most errors (start-up errors are the exception) terminate the CodeView command under which the error occurred, but do not terminate the debugger. You may see any of the following error messages:

`Bad address`

You specified the address in an invalid form. For example, you may have entered an address containing hexadecimal characters when the radix is decimal.

`Bad breakpoint command`

You typed an invalid breakpoint number with the Breakpoint Clear, Breakpoint Disable, or Breakpoint Enable command. The number must be in the range 0–19.

`Bad flag`

You specified an invalid flag mnemonic with the Register dialog command (**R**). Use one of the mnemonics displayed when you enter the command **RF**.

`Bad format string`

You specified an invalid type specifier following an expression. Expressions used with the Display Expression, Watch, Watchpoint, and Tracepoint commands can have **printf** type specifiers set off from the expression by a comma. The valid type specifiers are **d**, **i**, **u**, **o**, **x**, **X**, **f**, **e**, **E**, **g**, **G**, **c**, and **s**. Some type specifiers can be preceded by the prefix **h** or **l**. See Chapter 6, "Examining Data and Expressions," for more information about type specifiers.

`Bad radix (use 8, 10, or 16)`

The CodeView debugger uses only octal, decimal, and hexadecimal radixes.

`Bad register`

You typed the Register command (**R**) with an invalid register name. Use **AX**, **BX**, **CX**, **DX**, **SP**, **BP**, **SI**, **DI**, **DS**, **ES**, **SS**, **CS**, **IP**, or **F**.

`Bad type cast`

The valid types for type casting are the C types **void**, **char**, **int**, **short**, **long**, **signed**, **unsigned**, **float**, and **double**. The types **unsigned**, **signed**, **long**, and **short** can be combined with other types (**unsigned char**, for example) as listed in the *Microsoft C Compiler Language Reference*.

`Bad type (use one of 'ABDILSTUW')`

The valid dump types are ASCII (**A**), Byte (**B**), Integer (**I**), Unsigned (**U**), Word (**W**), Double Word (**D**), Short Real (**S**), Long Real (**L**), and 10-Byte Real (**T**).

`Badly formed type`

The type information in the symbol table of the file you are debugging is incorrect. If this message occurs, please note the circumstances of the error and inform Microsoft Corporation, using the Software Problem Report at the back of this manual.

`Breakpoint # or '*' expected`

You entered the Breakpoint Clear (**BC**), Breakpoint Disable (**BD**), or Breakpoint Enable (**BE**) command with no argument. These commands require that you specify the number of the breakpoint to be acted on, or that you specify the asterisk (*), indicating that all breakpoints are to be acted on.

`Cannot use struct or union as scalar`

A struct or union variable cannot be used as a scalar value in a C expression. Such variables must be followed by a file specifier or preceded with the address-of operator.

`Can't find` *filename*

The CodeView debugger could not find the executable file you specified when you started. You probably misspelled the file name, or the file is in a different directory.

Constant too big

The CodeView debugger cannot accept a constant number larger than 4,294,967,295 (0xFFFFFFFF).

Divide by zero

An expression in an argument of a dialog command attempts to divide by zero.

Expression too complex

An expression given as a dialog command argument is too complex. Simplify the expression.

Extra input ignored

You specified too many arguments to a command. The CodeView debugger evaluates the valid arguments and ignores the rest. Often, in this situation, the debugger will not evaluate the arguments the way you intended.

Floating point error

If this message occurs, please note the circumstances of the error and inform Microsoft Corporation, using the Software Problem Report at the back of this manual.

Internal debugger error

If this message occurs, please note the circumstances of the error and inform Microsoft Corporation, using the Software Problem Report at the back of this manual.

Invalid argument

One of the arguments you specified is not a valid CodeView expression.

Missing '"'

You specified a string as an argument to a dialog command, but you did not supply a closing double quotation mark.

Missing '('

An argument to a dialog command was specified as an expression containing a right parenthesis, but no left parenthesis.

`Missing ')'`

An argument to a dialog command was specified as an expression containing a left parenthesis, but no right parenthesis.

`Missing ']'`

An argument to a dialog command was specified as an expression containing a left bracket, but no right bracket. This error can also occur if a regular expression is specified with a right bracket but no left bracket.

`No closing single quote`

You specified a character in an expression used as a dialog command argument, but the closing single quotation mark is missing.

`No code at this line number`

You tried to set a breakpoint on a source line that does not correspond to code. For example, the line may be a data declaration or a comment.

`No match of regular expression`

No match was found for the regular expression you specified with the Search command or with the Find selection from the Search menu.

`No previous regular expression`

You selected Previous from the Search menu, but there was no previous match for the last regular expression specified.

`No program to debug`

You have executed to the end of the program you are debugging. You must restart the program (using the Restart command) before using any command that executes code.

`No source lines at this address`

The address you specified as an argument for the View command (**V**) does not have any source lines. For example, it could be an address in a library routine or an assembly-language module.

`No such file/directory`

A file you specified in a command argument or in response to a prompt does not exist. For example, the message appears when you select Load from the File menu, and then enter the name of a nonexistent file.

`No symbolic information`

The program file you specified is not in the CodeView format. You cannot debug in source mode, but you can use assembly mode.

`Not a text file`

You attempted to load a file using the Load selection from the File menu or using the View command, but the file is not a text file. The CodeView debugger determines if a file is a text file by checking the first 128 bytes for characters that are not in the ASCII ranges 9–13 and 20–126.

`Not an executable file`

The file you specified to be debugged when you started the CodeView debugger is not an executable file having the extension **.EXE** or **.COM**.

`Not enough space`

You typed the Shell Escape command (!) or selected Shell from the File menu, but there is not enough free memory to execute **COMMAND.COM**. Since memory is released by code in the C start-up routines, this error always occurs if you try to use the Shell Escape command before you have executed any code. Use any of the code-execution commands (Trace, Program Step, or Go) to execute the C start-up code, then try the Shell Escape command again. The message also occurs with assembly-language programs that do not specifically release memory.

`Object too big`

You entered a Tracepoint command with a data object (such as an array) that is larger than 128 bytes. You can watch data objects larger than 128 bytes using the Tracepoint Memory command.

`Operand types incorrect for this operation`

An operand in a C expression had a type incompatible with the operation applied to it. For example, if `p` is declared as `char *`, then `? p*p` would produce this error, since a pointer cannot be multiplied by a pointer.

`Operator must have a struct/union type`

You used the one of the member-selection operators (-> or .) in an expression that does not reference an element of a structure or a union.

`Operator needs lvalue`

You specified an expression that does not evaluate to an lvalue in an operation that requires an lvalue. For example, `? 3=100` is invalid. See the *Microsoft C Compiler Language Reference* for more information on lvalues.

`Program terminated normally` (*number*)

You executed your program to the end. The number displayed in parentheses is the exit code returned to MS-DOS by your program. You must use the Restart command (or the Start menu selection) to start the program before executing more code.

`Register variable out of scope`

You tried to specify a register variable using the period (.) operator and a function name. For example, if you are in a third-level function, you can display the value of a local variable called `local` in a second-level function called `parent` with the following command:

```
? parent.local
```

However, this command will not work if `local` is declared as a register variable.

`Regular expression too complex`

The regular expression specified is too complex for the CodeView debugger to evaluate.

`Regular expression too long`

The regular expression specified is too long for the CodeView debugger to evaluate.

`Syntax error`

You specified an invalid command line for a dialog command. Check for an invalid command letter. This message also appears if you enter an invalid assembly-language instruction using the Assemble command. The error will be preceded by a caret that points to the first character the CodeView debugger could not interpret.

`Too many breakpoints`

You tried to specify a 21st breakpoint. The CodeView debugger permits only 20.

`Too many open files`

You do not have enough file handles for the CodeView debugger to operate correctly. You must specify more files in your **CONFIG.SYS** file. See your MS-DOS user's guide for information on using the **CONFIG.SYS** file.

`Type conversion too complex`

You tried to type cast an element of an expression in a type other than the simple types or with more than one level of indirection. An example of a complex type would be type casting to a struct or union type. An example of two levels of indirection would be `char **`.

`Unable to open file`

A file you specified in a command argument or in response to a prompt cannot be opened. For example, the message appears when you select Load from the File menu, and then enter the name of a file that is corrupted or has its file attributes set so that it cannot be opened.

`Unknown symbol`

You specified an identifier that is not in the CodeView debugger's symbol table. Check for a misspelling. A symbol name spelled with letters of the wrong case will not be recognized unless the Case Sense selection on the Options menu has been turned off. This message may also occur if you try to use a local variable in an argument when you are not in the function where the variable is defined.

`Unrecognized option` *option*
`Valid options: /B /C<command> /F /S /T /W`

You entered an invalid option when starting the CodeView debugger. Retype the command line.

`Usage: cv [options] file [arguments]`

You failed to specify an executable file when you started the CodeView debugger. Try again with the syntax shown in the message.

`Video mode changed without /S option`

The program changed video modes (from or to one of the graphics modes) when screen swapping was not specified. You must use the **/S** option to specify screen swapping when debugging graphics programs. You can continue debugging when you get this message, but the output screen of the debugged program may be damaged.

`Warning: packed file`

You started the CodeView debugger with a packed file as the executable file. You can attempt to debug the program in assembly mode, but the packing routines at the start of the program may make this difficult. You cannot debug in source mode because all symbolic information is stripped from a file when it is packed with the **/EXEPACK** linker option or the **EXEPACK** utility.

Glossary

Address

A C expression that evaluates to an address in memory. Addresses can be given in the *segment:offset* format. If the *segment* is not given, the default segment is assumed. The default segment is **CS** for commands related to code and **DS** for commands related to data.

Address range

A range of memory bounded by two addresses. The range can be specified in the normal format by giving the starting and ending addresses (inclusive), or it can be specified in the object-range format by specifying the starting address followed first by the letter "L" (uppercase or lowercase) and then by the number of objects in the range (0x100 L 10, for example, specifies the range from 0x100 to 0x109, inclusive).

Assembly mode

The mode in which the CodeView debugger displays assembly-language-instruction mnemonics to represent the code being executed.

Basic input/output system (BIOS)

The code built into system memory that provides the lowest level of functionality in a computer system. You can trace into the BIOS with the CodeView debugger, using assembly mode.

Breakpoint

A specified address where program execution will be halted. The CodeView debugger stops whenever it reaches an address where a breakpoint has been set. *See* "Watchpoint" and "Tracepoint" for a description of conditional breakpoints.

Click

To press and quickly release one of the buttons on the mouse while pointing the mouse at an object on the screen.

Color graphics adapter (CGA)

A video adapter capable of displaying text characters or graphics pixels. Color can also be displayed with the appropriate display monitor.

Cursor

The thin blinking line that represents the current location where the CodeView debugger is ready to accept commands. The cursor need not be in the dialog window to enter dialog commands.

Dialog commands

Commands entered in the dialog window in window mode, or any command in sequential mode. Dialog commands consist of one- or two-character commands that can usually be followed by arguments.

Dialog window

The window at the bottom of the CodeView screen where dialog commands can be entered and previously entered dialog commands can be reviewed.

Dialog box

A box that appears when you select a menu item that requires a response. The box asks you to enter some text. After you type your response and press the ENTER key (or a mouse button), the box disappears.

Display window

The window above the dialog window where source code is displayed in source mode or assembly-language instructions are displayed in assembly mode.

Drag

To point the mouse at an object on the screen and press a mouse button, then while holding the button down, move the mouse. The object will move in the direction of the mouse movement. When the item is where you want it, release the button. The object will stay at that point.

Dump

To display the contents of memory at a specified memory location. In the CodeView debugger, the size of the object to be displayed is specified with a type character from the following list: **A** (ASCII), **B** (Byte), **I** (Integer), **U** (Unsigned Integer), **W** (Word), **D** (Double Word), **S** (Short Real), **L** (Long Real), or **T** (10-Byte Real).

Enhanced graphics adapter (EGA)

A video adapter capable of displaying in all the modes of the color graphics adapter (CGA) plus many additional modes. The CodeView **/43** option displays in the EGA's 43-line text mode.

Flags

A register that controls the operation of many machine-level instructions. In other registers, the contents of the register are considered as a whole, while in the flags register only the individual bits have meaning. In the CodeView debugger, the current values of the most commonly used bits of the flags register are shown at the bottom of the register window. *See also* "Registers."

Flipping

A screen-exchange method that uses the video pages of the CGA or EGA to store both the debugging and output screens. Video pages are areas of memory reserved for screen storage. When you request the other screen, the two video pages are exchanged. This method is faster than swapping, the other screen-exchange method, but it does not work with the MA or with programs that do graphics or use the video pages. *See also* "Screen exchange" and "Swapping."

Function call

A call to a a subroutine that performs a specific action. In C (source mode), subroutines are called functions. In assembly language (assembly mode), subroutines are called procedures.

Global symbol

A symbol that is available throughout the entire program. In the CodeView debugger, function names are always global symbols. *See also* "Local symbol."

Highlight

The reverse-video line that marks the current selection on a menu. The highlight moves as you move the mouse or press the UP ARROW or DOWN ARROW key.

Identifier

A name that identifies a register or a location in memory. The terms "identifier" and "symbol" are used synonymously in CodeView documentation.

Interrupt call

A machine-level procedure that can be called to execute a BIOS, MS-DOS, or other function. You can trace into BIOS interrupts with the CodeView debugger, but not into the MS-DOS interrupt (0x21).

Label

A symbol (identifier) representing an address in the code segment (**CS**) register. Labels in C programs can be either function names or labels for **goto** statements.

Local symbol

A symbol that only has value within a particular function. A function argument or a variable declared as **auto** or **static** within a function can be a local symbol. *See also* "Global symbol."

Lvalue

An expression that refers to a memory location. For example, `sym` or `buffer[count]` could be lvalues since they represent symbols in memory. However, `i==10` is not an lvalue because it evaluates to either 1 (true) or 0 (false) rather than a value stored in memory. Similarly, `sym1+sym2` could not be an lvalue because it refers to the sum of two variables rather than a single memory location. See the *Microsoft C Compiler Language Reference* for more information on lvalue expressions.

Menu bar

The bar at the top of the CodeView display containing menu titles and the titles Trace! and Go!.

Message box

A box that pops up and displays a message. In window mode, message boxes are used to display error messages. You can press any key or a mouse button to make the box disappear.

Monochrome adapter (MA)

A video adapter capable of displaying only in black and white. Most monochrome adapters display text only; individual graphics pixels cannot be displayed. The CodeView debugger recognizes monochrome adapters and automatically selects swapping as the screen-exchange mode.

Mouse

A small pointing device designed to fit comfortably under your hand. By moving it about on a flat surface, you can move the mouse pointer in the corresponding direction on the screen.

Object range

See "Address range."

Output screen

The screen where program output is shown. The Screen Exchange command (\), Output from the View menu, and the F4 key can be used to switch to this screen. The output screen is the same as it would be if you ran the debugged program outside of the CodeView debugger.

Pointer

The reverse-video square that moves to indicate the current position of the mouse. The mouse pointer only appears if a mouse is installed. To select an item with the mouse, move the mouse until the pointer rests on the item.

Popup menu

A menu that pops up when you point the mouse cursor to the menu title and press a mouse button. In the CodeView debugger, popup menus also pop up when you press the ALT key and the first letter of the menu title at the same time. You can make a selection from the menu by dragging the highlight up or down with the mouse, by pressing the UP ARROW or DOWN ARROW key to move the highlight, or by pressing the ALT key and the first letter of the selection title at the same time.

printf

A function in the C standard library that prints formatted output according to instructions supplied with a type-specifier argument. The CodeView debugger uses a subset of the **printf** type specifiers to format expression values.

Procedure call

A call to a subroutine that performs a specific action. In assembly language (assembly mode), subroutines are called procedures. In C (source mode), subroutines are called functions.

Program Step

To trace the next source line in source mode, or the next instruction in assembly mode. If the source line or instruction contains a function, procedure, or interrupt call, the call is executed to the end and the CodeView debugger is ready to execute the instruction after the call. *See also* "Trace."

Radix

The number system in which numbers are specified. In the CodeView debugger, numbers can be entered in three radixes: 8 (octal), 10 (decimal), or 16 (hexadecimal). The default radix is 10.

Redirection

To specify the device from which a program will receive input or to which it will send output. Normally program input comes from the keyboard, and program output goes to the screen. Redirection involves specifying a device (or file) other than the default device. In the MS-DOS operating system, input is redirected with the less-than symbol (<) and output is redirected with the greater-than symbol (>). The same symbols are used in the CodeView debugger to redirect input or output of the debugging session. In addition, the equal sign (=) can be used to redirect both input and output.

Registers

The places in memory where byte- or word-sized data can be stored during machine-level processing. The registers used with the 8086 family of processors are: **AX**, **BX**, **CX**, **DX**, **SP**, **BP**, **SI**, **DI**, **DS**, **ES**, **SS**, **CS**, **IP**, and the flags register. *See also* "Flags."

Register window

The optional window in which the central processing unit (CPU) registers and the bits of the flag register are displayed.

Regular expressions

A system of specifying text patterns that match variable text strings. The CodeView debugger supports a subset of the regular-expression characters used in the XENIX and UNIX operating systems. Regular expressions can be used to find strings in source files.

Screen exchange

The method by which both the output screen and the debugging screen are kept in memory so that both can be updated simultaneously and

either viewed at the user's convenience. The two screen-exchange modes are flipping and swapping. *See also* "Flipping" and "Swapping."

Sequential mode

The mode in which all CodeView output is sequential and no windows are available. Input and output scroll down the screen and the old output scrolls off the top of the screen when the screen is full. You cannot examine previous commands after they scroll off the top. This mode is required with computers that are not IBM compatible. The mouse and most window commands are not supported in sequential mode. Any debugging operation that can be done in window mode can also be done in sequential mode.

Shell escape

A method of leaving the CodeView debugger without losing the current debugging context. You can "escape to a shel!," do various MS-DOS tasks, and then return to the debugger. The debugging screen will be the same as when you left it. The CodeView debugger creates the shell by saving all current operations to memory and invoking a second copy of **COMMAND.COM**.

Source mode

The mode in which the CodeView debugger displays C source code to represent the code being executed.

Stack trace

A symbolic representation of the functions that have been executed to reach the current instruction address. As a function is executed, the function address and any function arguments are pushed on the stack (the area of memory starting at the address of the **SS** register). Therefore, a trace of the stack always shows the currently active functions and the values of their arguments.

Start-up code

The code that the C compiler places at the beginning of every program to control execution of the program code. When the CodeView debugger is loaded, the first source line executed runs the entire start-up code. If you switch to assembly mode before executing any code, you can trace through the start-up code.

Swapping

A screen-exchange method that uses buffers to store the debugging and output screens. When you request the other screen, the two buffers are exchanged. This method is slower than flipping, the other screen-exchange method, but it works with any adapter and any type of program. *See also* "Flipping" and "Screen Exchange."

Symbol

A name that identifies a location in memory. The terms "symbol" and "identifier" are used synonymously in CodeView documentation.

Toggle

A function key or menu selection that turns a feature off if it is on, or on if it is off. When used as a verb, toggle means to reverse the status of a feature. For example, the F3 key is a toggle that switches between source and assembly modes. You can press the F3 key to toggle between the two modes.

Trace

To trace the next source line in source mode, or the next instruction in assembly mode. If the source line or instruction contains a function, procedure, or interrupt call, the first source line or instruction of the call is executed. The CodeView debugger is ready to execute the next instruction inside the call. *See also* "Program Step."

Tracepoint

A variable breakpoint that is taken when a specified value changes. The value to be tested can be either the value of a CodeView expression, or any of the values in a range of memory. Tracepoints can slow program execution significantly, since the CodeView debugger has to check after executing each source line in source mode or after each instruction in assembly mode to see if the value has changed. *See also* "Breakpoint."

Type casting

To specify a type specifier in parentheses preceding an expression to indicate the type of the expression's value. For example, if `x` and `y` are integer values with the values 5 and 2 respectively, the expression `x/y` indicates integer division and the expression has the value 2. The expression `(float)x/y` indicates real-number division and has the value 2.5.

Watch window

The window where watch statements and their values are displayed. The three kinds of watch statements are watch expressions, watchpoints, and tracepoints.

Watchpoint

A variable breakpoint that is taken when a specified expression becomes nonzero (true). Watchpoints can slow program execution significantly, since the CodeView debugger has to check after executing each source line in source mode or after each instruction in assembly mode to see if the value is true. *See also* "Breakpoint."

Window commands

Commands that work only in the CodeView debugger's window mode. Window commands consist of function keys, mouse selection, CONTROL and ALT key combinations, and selections from popup menus.

Window mode

The mode in which the CodeView debugger displays separate windows, which can change independently. The debugger has mouse support and a wide variety of window commands in window mode.

Index

CodeView Index

! (exclamation point)
  command indicator, 37
  Shell Escape command, 197, 217, 233
" (quotation mark), Pause command, 204, 216
(number sign), Tab Set command, 198, 217
$ (dollar sign), in regular expressions, 227
* (asterisk)
  Comment command, 202, 216
  regular expressions, used in, 226
– (dash)
  option designator, 23
  regular expressions, used in, 225
. (period)
  Current Location command, 161, 217
  operator, 71, 234
  regular expressions, used in, 224
/ (slash)
  option designator, 23
  Search command, 194, 217, 234
: (colon)
  Delay command, 204, 216
  operator, 71, 75
< (less-than sign), Redirected Input command, 199, 217
= (equal sign), Redirected Input and Output command, 201, 217
> (CodeView prompt), 37, 38, 67
> (greater-than sign), Redirected Output command, 200, 217
? (question mark), Display Expression command, 216
@ (at sign)
  Redraw command, 191, 217
  register prefix, 74
[ ] (brackets)
  notational conventions, 10
  regular expressions, used in, 225
\ (backslash), Screen Exchange command, 192, 211, 217, 241
^ (caret), in regular expressions, 226, 227
_ (underscore), in symbol names, 72
| (vertical bar), notational conventions, 10

10-byte reals
  dumping, 112
  entering, 180
/43 CodeView option, 31, 212, 239
7, 8087 command, 116, 218
8087
  command, 115, 216
  coprocessor, 115, 170

A (Assemble command), 168, 216
Absolute addresses, 75
Address ranges, arguments as, 76, 237
Addresses
  absolute, 135
  arguments, used as, 75, 229, 232, 237
  full, 75, 134
Arguments
  dialog commands, 67, 69, 231, 234
  function, 61, 162, 243
  program, 21, 92
ASCII characters, 106, 107
Assemble command, 167, 216, 234
Assembly
  address, 168
  language, 6
  mode, 32, 57, 153, 211–212, 233, 237
  rules, 168
Asterisk (*)
  Comment command, 202, 216
  regular expressions, used in, 226
At sign (@)
  Redraw command, 191, 217
  register prefix, 74

/B CodeView option, 11, 25, 212
Backslash (\), Screen Exchange command, 192, 211, 217, 241
BACKSPACE key, 68
Basic input/output system (BIOS), 237

BC (Breakpoint Clear command), 125, 216, 229, 230
BD (Breakpoint Disable command), 126, 216, 229, 230
BE (Breakpoint Enable command), 127, 216, 229, 230
BIOS (basic input/output system), 237
BL (Breakpoint List command), 128, 216
Black-and-white display, 11, 25, 212
Bold type, notational conventions, 8
BP (Breakpoint Set command), 122, 216
Brackets ([ ])
  notational conventions, 10
  regular expressions, 225
Breakpoint Clear command, 55, 124, 214, 216, 229, 230
Breakpoint Disable command, 125, 216, 229, 230
Breakpoint Enable command, 127, 216, 229, 230
Breakpoint List command, 128, 216
Breakpoint Set command, 41, 45, 64, 121, 213, 216, 232, 234
Breakpoint
  address, 88
  defined, 121, 237
  deletion, 124, 216
  display of, 37, 122
  Go command, used with, 87
  listing, 128, 216
Buffer, command, 39, 68

/C CodeView option, 26, 212
C expressions, 70, 231
C operators, 70
Calling conventions, 162
Calls menu, 60, 163, 214
Capital letters, notational conventions, 8, 10
Caret (^), in regular expressions, 226, 227
Case sensitivity, 60, 69, 72, 214, 235
CGA (color graphics adapter), 237
CL compiler control program, 16, 17, 19
Click, defined, 44, 237
/CODEVIEW link option, 18, 19, 32

Colon (:)
  Delay command, 204, 216
  operator, 71, 75
Color graphics adapter (CGA), 28, 31, 237
.COM extension, for debugged files, 20, 32, 233
Command buffer, 39, 68
COMMAND.COM, with Shell command, 50, 195
Commands
  8087, 115, 218
  Assemble, 167, 216, 234
  Breakpoint Clear, 55, 124, 214, 216, 229, 230
  Breakpoint Disable, 125, 216, 229, 230
  Breakpoint Enable, 127, 216, 229, 230
  Breakpoint List, 128, 216
  Breakpoint Set, 41, 45, 64, 121, 213, 216, 232, 234
  Comment, 202, 216
  Current Location, 161, 217
  Delay, 204, 216
  Dialog, 37, 38, 67, 238
  Display Expression, 54, 95, 214, 216
  Dump
    10-Byte Reals, 112
    ASCII, 106
    Bytes, 106
    default size, 104, 105
    Double Words, 109
    Integers, 107
    Long Reals, 111
    Short Reals, 110
    summary, 216
    Unsigned Integers, 108
    Words, 109
  Enter, 174
    10-Byte Reals, 180
    ASCII, 175
    Bytes, 175
    Double Words, 178
    Integers, 176
    Long Reals, 180
    Short Reals, 179
    summary, 216
    Unsigned Integers, 177
    Words, 177
  Examine Symbols, 100, 216

Commands *(continued)*
Execute, 55, 90, 214, 216
Expression, 54, 95, 214, 216
Go, 41, 47, 63, 87, 213, 216
Goto, 41, 46, 87, 213, 216
Help, 40, 53, 61, 63, 187, 211, 213, 214, 216
move cursor down, 39, 213
move cursor up, 39, 213
move separator line down, 39, 44, 213
move separator line up, 39, 44, 213
Pause, 204, 216
Program Step, 41, 46, 64, 84, 213, 217, 242
Quit, 50, 188, 214, 217
Radix, 73, 189, 217, 229, 242
Redirected Input, 199, 217
Redirected Input and Output, 201, 217
Redirected Output, 200, 217
Redirection, 199, 217, 242
Redraw, 191, 217
Register, 40, 47, 59, 63, 113, 181, 213, 217, 229, 230, 242
Restart, 55, 91, 214, 217, 232, 234
Screen Exchange, 40, 53, 63, 191, 211, 213, 214, 217, 241, 242
scroll line down, 45
scroll line up, 45
scroll page down, 39, 45, 213
scroll page up, 39, 45, 213
scroll to bottom, 40, 45, 213
scroll to top, 39, 45, 213
Search, 51, 192, 214, 217, 223, 232
Set Mode, 40, 53, 63, 153, 211, 213, 214, 217
Shell Escape, 50, 195, 214, 217, 233, 243
Stack Trace, 61, 162, 214, 217, 243
Tab Set, 198, 217
Trace, 41, 46, 64, 82, 213, 217, 244
Tracepoint..., 56, 64, 141, 214, 218, 229, 233, 244
Unassemble, 155, 217
View, 158, 217, 232, 233
Watch, 56, 64, 134, 214, 218, 229
Watch Delete, 57, 146, 214, 218
Watch List, 64, 148, 218
Watchpoint..., 56, 64, 138, 214, 218, 229, 245
Comment
command, 202, 216
line, 87, 88, 122, 123
Compiler errors, 18
COMSPEC environment variable, 195
Conditional breakpoints, 56, 121, 133
CONFIG.SYS file, 235
Constant numbers, as arguments, 73, 231
CONTROL-BREAK, 42, 82, 141
CONTROL-C, 42, 67, 82, 141
CONTROL-D, 39, 213
CONTROL-S, 68
CONTROL-U, 39, 213
/CPARMAXALLOC link option, 196
Current Location command, 161, 217
Current location line, 37
Cursor, 37, 67, 238
CV.EXE, location of, 19
CV.HLP, location of, 19, 61

D (Dump command), 105
DA (Dump ASCII command), 106
Dash (–)
option designator, 23
regular expressions, 225
DB (Dump Bytes command), 106
DD (Dump Double Words command), 109
DEBUG, 6, 35, 63
Debugging modes, 211
Default
address-range size, 104
assembly-mode format, 57
expression format, 136
IBM Personal Computer, used with, 23
radix, 162, 189, 190, 242
segment, 75
start-up behavior, 21
type, 105, 136, 144, 174, 215
Delay command, 204, 216
Destination address, with Go command, 87
DI (Dump Integers command), 107
Dialog box, 38, 43, 48, 238
Dialog window, 37, 238
Display
mode, 81, 156, 159, 211
window, 37, 238

Display Expression command, 54, 95, 214, 216
Divide by zero, 231
DL (Dump Long Reals command), 111
Dollar sign ($), in regular expressions, 227
DOWN ARROW key (cursor down), 39, 213
Drag, defined, 44, 238
DS (Dump Short Reals command), 110
DT (Dump 10-Byte Reals command), 112
DU (Dump Unsigned Integers command), 108
Dump address, 104
Dump commands, 104
  10-Byte Reals, 112
  ASCII, 106
  Bytes, 106
  default size, 105
  Double Words, 109
  Integers, 107
  Long Reals, 111
  Short Reals, 110
  summary, 216
Dump command
  Unsigned Integers, 108
  Words, 109
Dump, defined, 238
DW (Dump Words command), 109

E (Enter command), 174
E (Execute command), 90, 216
EA (Enter ASCII command), 175
EA (Enter Bytes command), 175
Echo, with redirection, 200
ED (Enter Double Words command), 178
EGA (enhanced graphics adapter), 239
EI (Enter Integers command), 176
EL (Enter Long Reals command), 180
Ellipses, notational conventions, 9
END key (exit help), 62
END key (scroll to bottom), 40, 213
Enhanced graphics adapter (EGA), 28, 31, 239
Enter commands
  10-Byte Reals, 180
  ASCII, 175
  Bytes, 175
Enter commands *(continued)*
  default size, 174
  described, 170, 216
  Double Words, 178
  Integers, 176
  Long Reals, 180
  Short Reals, 179
  Unsigned Integers, 177
  Words, 177
Equal sign (=), Redirected Input and Output command, 201, 217
Error
  internal debugger, 230, 231
  messages, 227
Errorlevel code, 87
Errors, logical, 18
ES (Enter Short Reals command), 179
ESCAPE key, 43
ET (Enter 10-Byte Reals command), 180
EU (Enter Unsigned Integers command), 177
Evaluate, menu selection, 97
EW (Enter Words command), 177
Examine Symbols command, 100, 216
Examples, notational conventions, 9
Exchange modes, 211–212
Exclamation point (!)
  command indicator, 37
  Shell Escape command, 197, 217, 233
.EXE extension, for debugged files, 20, 21, 32, 233
Executable file
  CodeView format, 16, 18
  command line, used in, 20, 230, 233
  location of, 20
  required for start-up, 21
Execute command, 55, 90, 214, 216
/EXEPACK linker option, 18, 236
EXEPACK utility, 18
Exit code, 87, 89
Exit, MS-DOS command, 50, 196
Expression evaluation, 54, 70, 95, 214, 216, 231
Expressions, regular, 51, 192, 223, 232, 234, 242

/F CodeView option, 27, 212
F1 key (Help), 40, 61, 63, 188, 211, 213
F10 key (Program Step), 41, 64, 85, 213

F2 key (Register), 40, 63, 113, 181, 213
F3 key (Set source/assembly), 40, 63, 154, 211, 213
F4 key (Screen Exchange), 40, 63, 192, 211, 213, 241
F5 key (Go), 41, 63, 88, 213
F6 key (switch cursor), 39, 88, 213
F7 key (Goto), 41, 88, 213
F8 key (Trace), 41, 64, 83, 213
F9 key (Breakpoint Clear), 124
F9 key (Breakpoint Set), 41, 64, 127, 213
Far-return mnemonic (RETF), 168
File handles, 235
File menu
  Load..., 49, 158, 214, 232, 233, 235
  Quit, 50, 188, 214
  Shell, 50, 196, 214, 233, 243
Flag bits, 47, 114, 181, 227, 239
Flag mnemonics, 182, 229
Flipping, screen, 27, 212, 239
Function
  calls, 61, 82, 85, 163, 239, 243
  keys
    CONTROL-D (separator line down), 39, 213
    CONTROL-U (separator line up), 39, 213
    DOWN ARROW (cursor down), 39, 213
    END (exit help), 62
    END (scroll to bottom), 40, 213
    F1 (Help), 40, 61, 63, 188, 209, 213
    F10 (Program Step), 41, 64, 85, 213
    F2 (Register), 40, 63, 113, 181, 213
    F3 (Set source/assembly), 40, 63, 154, 211, 213
    F4 (Screen Exchange), 40, 63, 192, 211, 213, 241
    F5 (Go), 41, 63, 88, 213
    F6 (switch cursor), 39, 88, 213
    F7 (Goto), 41, 88, 213
    F8 (Trace), 41, 64, 83, 213
    F9 (Breakpoint Clear), 124
    F9 (Breakpoint Set), 41, 64, 127, 213
    HOME (scroll to top), 39, 213
    HOME (top of help), 62
    PGDN (next help), 62
    PGDN (scroll page down), 39, 160, 213
  keys *(continued)*
    PGUP (previous help), 62
    PGUP (scroll page up), 39, 213
    UP ARROW (cursor up), 39, 213
Functions, 60, 100, 162

G (Go command), 88, 216
Global symbol, 239
Go command, 41, 47, 63, 87, 213, 216
Goto command, 41, 46, 87, 213, 216
Graphics programs, debugging, 201
Greater-than sign (>), Redirected Output command, 200, 217

H (Help command), 188, 211, 216
Help command, 40, 53, 61, 63, 187, 211, 213, 214, 216
Highlight, 38, 239
HOME key (scroll to top), 39, 213
HOME key (top of help), 62

IBM PC, CodeView compatibility with, 5, 28, 29
IBM PC, recognition, 23
Identifiers
  arguments, used as, 239, 244
  arguments, used in, 72, 235
Immediate operand, 169
Include files, 17
IND (indefinite), 105
Indentation, 198
Indirect register instructions, 169
Indirection levels, 71
INF (infinity), 105
Infinity, 105
Instruction, current, 82, 84
Instruction-name synonyms, 169
Integers, dumping, 107
Interrupt
  21 (MS-DOS functions), 82
  calls, 240
Italic type, notational conventions, 9

K (Stack Trace command), 163, 217
Key names, notational conventions, 10

L (Restart command), 92, 217
Labels
  defined, 240
  finding, 52, 194, 214
Less-than sign (<), Redirected Input command, 199, 217
Line numbers, as arguments, 77
LINK (object linker), 16, 19
Load, menu selection, 92
Long reals
  dumping, 111
  entering, 180
Loops
  tracepoints, used with, 146
  watchpoints, used with, 141
Lvalue, 142, 240

/M CodeView option, 30, 212
MA (monochrome adapter), 240
Macro Assembler, 6, 32
Macros, 17
Member-selection operators, 233
Memory release, 196, 233
Menu
  Calls, 60, 163, 214
  defined, 37
  File
    Load..., 49, 158, 214, 232, 233, 234
    Quit, 50, 188, 214
    Shell, 50, 196, 214, 233, 243
  keyboard selection from, 42
  mouse selection from, 47
  Options
    Bytes Coded, 59, 214
    Case Sense, 60, 214, 235
    Flip/Swap, 58, 214
    Mixed Source, 59, 154, 214
    Registers, 59, 113, 181, 214
    Symbols, 59, 154, 214
  Run
    Clear Breakpoints, 55, 124, 214
    Execute, 55, 90, 214
    Restart, 55, 91, 214, 234
    Start, 54, 91, 214, 234
  Search
    Find..., 51, 193, 214
    Label..., 52, 194, 214
    Next, 52, 193, 214
    Previous, 52, 193, 214
Menu *(continued)*
  View
    Assembly, 53, 154, 211–212, 214
    Evaluate..., 54, 97, 214
    Help, 53, 61, 187, 211, 214
    Output, 53, 192, 211, 214, 241
    Source, 53, 154, 211, 214
  Watch
    Add Watch..., 56, 135, 214
    Delete Watch..., 57, 146, 214
    Tracepoint..., 56, 143, 214
    Watchpoint..., 56, 139, 214
Menu bar, 37, 240
Menu selection, Start, 91
Message box, 38, 43, 48, 240
Mixed mode, 153, 211–212
Modules, examination, 100
Monochrome adapter (MA), 28, 31, 240
Mouse
  compatibility, 6
  defined, 241
  driver, 31
  ignore option, 30, 212
  pointer, 38, 44
  selection with, 44
MSC compiler control program, 16, 17, 18

N (Radix command), 190, 217, 227
NAN (not a number), 105
Notational conventions, 9
  bold type, 8
  brackets, 10
  capital letters, 8
  ellipses, 9
  examples, 9
  italic type, 9
  quotation marks, 10
  sample screens, 11
  vertical bar, 10
Number sign (#), Tab Set command, 198, 217
Numbers
  arguments, used as, 73, 231
  floating point, 110, 111, 112

Object ranges, as arguments, 76, 241
/Od compiler option, 17
Operands, 114

Operators, 70
Optimization, 17
Optional fields, conventions for, 10
Options menu
  Bytes Coded, 59, 214
  Case Sense, 60, 214, 235
  Flip/Swap, 58, 214
  Mix Source, 59, 214
  Mixed Source, 154
  Registers, 59, 113, 181, 214
  Symbols, 59, 214
Options
  CodeView
    /43, 31, 212, 239
    /B, 25, 212
    /C, 26, 212
    command line, used in, 20
    /F, 27, 212
    /M, 30, 212
    /S, 27, 212, 235
    summary, 23
    /T, 29, 211, 212
    /W, 29, 211, 212
  compiler
    /Od, 17
    /Zd, 17
    /Zi, 17
  linker
    /CODEVIEW, 18, 19, 32
    /CPARMAXALLOC, 196
    /EXEPACK, 18, 236
Output screen, 27, 191, 241, 242
Overlays, 20

P (Program Step command), 85, 217
Parameters, program, 21
Pass count, 122, 129
PATH command, 19
Pause command, 204, 216
PC-DOS, 23
Period (.)
  Current Location command, used as, 161, 217
  operator, 71, 234
  regular expressions, used in, 224
PGDN (next help), 62
PGDN (scroll page down), 39, 160, 213
PGUP (previous help), 62
PGUP (scroll page up), 39, 213
Point, defined, 44
Pointer, mouse, 38, 44, 241
Popup menu, defined, 241
Precedence of operators, 70
Prefixes, with type specifiers, 97, 219, 229
printf type
  prefixes, 97, 217, 229, 241
  specifiers, 54, 95, 136, 139, 143, 218
Procedure calls, 82, 85, 241
Procedures, 100, 162
Program Step command, 41, 46, 64, 84, 213, 217, 242
Prompt (>), CodeView, 37, 38, 67
Protected-mode (80286) mnemonics, 155, 157, 167
Public symbols, 32

Q (Quit command), 189, 217
Question mark (?), as Display Expression command, 216
Quit command, 50, 188, 214, 217
Quotation marks ("), as Pause command, 204, 216
Quotation marks ("), notational conventions, 10

R (Register command), 114, 182, 217, 229, 230
Radix, 189
  command, 73, 189, 217, 229, 242
  current, 61, 73, 162
Ranges, as arguments, 76
README.DOC file, 8
Redirected Input and Output command, 201, 217
Redirected Input command, 199, 217
Redirected Output command, 200, 217
Redirection
  commands, 199, 217, 242
  start-up commands, used in, 26
Redraw command, 191, 217
Register command, 40, 47, 59, 63, 113, 213, 217, 229, 230, 242
  variables, 71, 142, 234
  window, 37, 242
Registers, argument, used as, 74
Regular expressions, 51, 52, 192, 193, 223, 232, 234, 242
Relational expressions, 138

Restart command, 55, 91, 214, 217, 232, 234
ROM (read-only memory), 83
Run menu
  Clear Breakpoints, 55, 124, 214
  Execute, 55, 90, 214
  Restart, 55, 91, 214, 234
  Start, 54, 91, 214, 234

/S CodeView option, 27, 212, 235
S (Set Mode command), 154, 211, 217
Screen
  buffer, 135
  modes, 211
Screen Exchange command, 40, 53, 63, 191, 211, 213, 214, 217, 241, 242
Screen exchange method, 27
Screen movement commands, 39, 213
Screens notational conventions, 11
Scroll bar, defined, 37
Search
  command, 51, 192, 214, 217, 223, 232
  menu
    Find..., 51, 193, 214
    Label..., 52, 193, 214
    Next, 52, 193, 214
    Previous, 52, 193, 214
Separator line, 37
Sequential mode, 29, 35, 63, 201, 211, 212, 243
Set Block, MS-DOS function call (0x4A), 196
Set Mode command, 40, 53, 63, 153, 211, 213, 214, 217
Shell Escape command, 50, 195, 214, 217, 233, 243
Short reals
  dumping, 110
  entering, 179
Slash (/) as
  option designator, 23
  Search Command, 194, 217, 234
Small capitals, notational conventions, 10
Source code
  writing, 16
  file, with line number arguments, 77
  mode, 153, 211, 233, 243
Source-module files, location of, 20, 50
Stack, 8087, 117
Stack Trace command, 61, 162, 214, 217, 243
Start-up
  code, 22, 50, 196, 243
  command line, 20, 230, 233
  commands, 26, 212
  file configuration, 19
String mnemonics, 168
Strings, as arguments, 78, 231
Style, of programming, 16
Swapping, screen, 27, 212, 244
Symbol names, spelling, 72
Symbols
  arguments, used in, 72, 235, 239
  definition of, 244
  examination, 100
SYMDEB, 6, 35, 63
Syntax
  errors, 18
  summary, 187, 216
Syntax conventions. *See* Notational conventions
SYSTEM-REQUEST key, 42, 82

/T CodeView option, 29, 211, 212
T (Trace command), 83, 217
Tab Set command, 198, 217
Text files, identification, 233
Text strings, finding, 51, 192, 214, 217, 223
Toggle, defined, 244
TP (Tracepoint command), 143, 218, 227
Trace command, 41, 46, 64, 82, 213, 217, 244
Tracepoint command, 56, 64, 141, 214, 218, 229, 233, 244
Tracepoint, defined, 141, 244
Two-color graphics display, 25, 210
Type casting, 99, 138, 230, 244
Type specifiers, 54, 95, 136, 139, 143, 218, 229

U (Unassemble command), 156, 218
Unassemble command, 155, 218
Underscore (_), in symbol names, 72
Unsigned integers, dumping, 108
UP ARROW key (cursor up), 39, 212

Uppercase letters, notational conventions, 8

V (View command), 159, 218 233
Variables, local, 17, 72, 134, 235, 240
Vertical bar, notational conventions, 10
Video-display pages, 27
Video modes, 235
View command, 158, 218, 232, 233
View menu
  Assembly, 53, 154, 211–212, 214
  Evaluate..., 54, 97, 214
  Help, 53, 61, 187, 211, 214
  Output, 53, 192, 211, 214, 241
  Source, 53, 154, 211, 214

/W CodeView option, 29, 211, 212
W (Watch command), 136, 218, 227
W (Watch List command), 148, 218
WAIT instruction, 170
Watch command, 56, 64, 134, 214, 218, 229
Watch Delete command, 57, 146, 214, 218
Watch-expression statement, 135
Watch List command, 64, 148, 218
Watch-memory statement, 136
Watch menu
  Add Watch..., 56, 135, 214
  Delete Watch..., 57, 146, 214
  Tracepoint..., 56, 143, 214
  Watchpoint..., 56, 139, 214, 245
Watch Statement commands, 37, 133
Watch statements
  deletion, 146, 218
  listing, 148, 218
Watch window, 37, 133
Watchpoint command, 56, 64, 138, 214, 218, 229, 245
Watchpoint, defined, 138, 218, 227, 245
Window commands, 38, 67, 245
Window mode, 29, 35, 211, 212, 245
WP (Watchpoint command), 139, 218, 227

X (Examine Symbols command), 101, 216

Y (Watch Delete command), 147, 218

/Zd compiler option, 17
Zero, division by, 231
/Zi compiler option, 17

Language Reference

Language Reference

Microsoft® C Compiler

Language Reference

Microsoft Corporation

Information in this document is subject to change without notice and does not represent a commitment on the part of Microsoft Corporation. The software described in this document is furnished under a license agreement or nondisclosure agreement. The software may be used or copied only in accordance with the terms of the agreement. It is against the law to copy this software on magnetic tape, disk, or any other medium for any purpose other than the purchaser's personal use.

© Copyright Microsoft Corporation, 1984, 1985, 1986

If you have comments about the software, complete the Software Problem Report at the back of this manual and return it to Microsoft Corporation.

If you have comments about the software documentation, complete the Documentation Feedback reply card at the back of this manual and return it to Microsoft Corporation.

Microsoft, the Microsoft logo, MS, MS-DOS, and XENIX are registered trademarks of Microsoft Corporation. CodeView and The High Performance Software are trademarks of Microsoft Corporation.

Document Number 410840018-400-R01-0486

Contents

1 Introduction 1

1.1 Overview 3
1.2 About This Manual 4
1.3 Notational Conventions 6

2 Elements of C 9

2.1 Introduction 11
2.2 Character Sets 11
2.3 Constants 17
2.4 Identifiers 22
2.5 Keywords 24
2.6 Comments 24
2.7 Tokens 25

3 Program Structure 27

3.1 Introduction 29
3.2 Source Program 29
3.3 Source Files 30
3.4 Program Execution 32
3.5 Lifetime and Visibility 33
3.6 Naming Classes 36

4 Declarations 39

4.1 Introduction 41
4.2 Type Specifiers 42
4.3 Declarators 46
4.4 Variable Declarations 53
4.5 Function Declarations 65
4.6 Storage Classes 68
4.7 Initialization 75
4.8 Type Declarations 80
4.9 Type Names 83

5 Expressions and Assignments 85

5.1 Introduction 87
5.2 Operands 87
5.3 Operators 96
5.4 Assignment Operators 116
5.5 Precedence and Order of Evaluation 120
5.6 Side Effects 123
5.7 Type Conversions 124

6 Statements 133

6.1 Introduction 135
6.2 The break Statement 137
6.3 The Compound Statement 138
6.4 The continue Statement 140
6.5 The do Statement 141
6.6 The Expression Statement 142
6.7 The for Statement 143
6.8 The goto and Labeled Statements 145
6.9 The if Statement 146
6.10 The Null Statement 148
6.11 The return Statement 149
6.12 The switch Statement 151
6.13 The while Statement 154

7 Functions 155

7.1 Introduction 157
7.2 Function Definitions 157
7.3 Function Declarations 164
7.4 Function Calls 166

8 Preprocessor Directives and Pragmas 175

8.1 Introduction 177
8.2 Manifest Constants and Macros 178
8.3 Include Files 183
8.4 Conditional Compilation 184
8.5 Line Control 189
8.6 Pragmas 190

Appendixes 191

A Differences 193

B Syntax Summary 199

B.1 Tokens 201
B.2 Expressions 205
B.3 Declarations 207
B.4 Statements 210
B.5 Definitions 211
B.6 Preprocessor Directives 211
B.7 Pragmas 212

Index 213

Tables

Table 2.1 Punctuation and Special Characters 13

Table 2.2 Escape Sequences 14

Table 2.3 Operators 16

Table 2.4 Examples of Integer Constants 18

Table 2.5 Types Assigned to Octal and Hexadecimal Constants 19

Table 2.6 Examples of Long Integer Constants 19

Table 2.7 Examples of Character Constants 21

Table 3.1 Summary of Lifetime and Visibility 35

Table 4.1 Fundamental Types 42

Table 4.2 Type Specifiers and Abbreviations 43

Table 4.3 Storage and Range of Values for Fundamental Types 44

Table 5.1 Precedence and Associativity of C Operators 120

Table 5.2 Conversions from Signed Integral Types 125

Table 5.3 Conversions from Unsigned Integral Types 126

Table 5.4 Conversions from Floating-Point Types 128

Chapter 1
Introduction

1.1 Overview 3
1.2 About This Manual 4
1.3 Notational Conventions 6

1.1 Overview

The C language is a general-purpose programming language well known for its efficiency, economy, and portability. While these advantages make it a good choice for almost any kind of programming, C has proved to be especially useful in systems programming because it allows programmers to write fast and compact programs and to transport those programs to other systems. In many cases, well-written C programs are comparable in speed to assembly-language programs, and they offer the advantages of easier maintenance and greater readability.

C combines efficiency and power in a relatively small language. C does not include built-in functions to perform tasks such as input and output, storage allocation, screen manipulation, and process control. Instead, C programmers rely on run-time libraries to perform such tasks.

This design contributes to C's adaptability and compactness. Because the language is relatively confined, it does not assume or impose a particular programming model. Run-time routines provide support as needed, allowing the programmer to minimize their use, if desired, or to tailor run-time routines for special purposes.

The design also helps to isolate language features from processor-specific features in a particular C implementation, thus aiding programmers who want to write portable code. The strict definition of the language makes it independent of any particular operating system or machine; at the same time, programmers can easily add system-specific routines to take advantage of a particular machine's efficiencies.

Some of the significant features of the C language are as follows:

- C provides a full set of loop, conditional, and transfer statements to control program flow logically and efficiently and to encourage structured programming.
- C offers an unusually large set of operators. Many of C's operators correspond to common machine instructions, allowing a direct translation into machine code. The variety of operators lets the programmer specify different kinds of operations clearly and with a minimum of code.
- C's data types include several sizes of integers, as well as single- and double-precision floating-point types. The programmer can design more complex data types, such as arrays and data structures, to suit specific program needs.

- C programmers can declare "pointers" to variables and functions. A pointer to an item corresponds to the machine address of that item. Using pointers wisely can increase program efficiency considerably, since pointers let the programmer refer to items in the same way the machine does. C also supports pointer arithmetic, allowing the programmer both to access and manipulate memory addresses directly.
- The C preprocessor, a text processor, acts on the text of files before compilation. Among its most useful applications for C programs are the definition of program constants, the substitution of function calls with faster macro look-alikes, and conditional compilation. The preprocessor is not limited to processing C files; it can be used on any text file.
- C is a flexible language, leaving much of the decision making up to the programmer. In keeping with this attitude, C imposes few restrictions in matters such as type conversion. While this is often an asset, C programmers must know the language well to understand how their programs will behave.

1.2 About This Manual

The *Microsoft® C Compiler Language Reference* defines the C language as implemented by Microsoft Corporation. It is intended as a reference for programmers who have experience in C or in another programming language. Knowledge of programming fundamentals is assumed.

Note

If you want a quick overview of how Microsoft C compares with the definition of C found in Appendix A of *The C Programming Language*, by Brian W. Kernighan and Dennis M. Ritchie, turn to Appendix A of this manual. Appendix B of this manual summarizes the syntax of the C language as defined by Microsoft.

The run-time library functions available for use in Microsoft C programs are discussed in a separate library reference manual.

Consult your system documentation for an explanation of how to compile and link C programs on your system. Your system documentation also contains information specific to the implementation of C on your system.

This manual is organized as follows:

Chapter 2, "Elements of C," describes the letters, numbers, and symbols that can be used in C programs and the combinations of characters that have special meanings to the C compiler.

Chapter 3, "Program Structure," discusses the components and structure of C programs and explains how C source files are organized.

Chapter 4, "Declarations," describes how to specify the attributes of C variables, functions, and user-defined types. C provides a number of predefined data types and allows the programmer to declare aggregate types and pointers.

Chapter 5, "Expressions and Assignments," describes the operands and operators that form C expressions and assignments. The chapter also discusses the type conversions and side effects that may accompany the evaluation of expressions.

Chapter 6, "Statements," describes C statements. Statements control the flow of program execution.

Chapter 7, "Functions," discusses features of C functions. In particular, this chapter explains how to define, declare, and call a function and describes function parameters and return values.

Chapter 8, "Preprocessor Directives and Pragmas," describes the instructions recognized by the C preprocessor. The C preprocessor is a text processor automatically invoked before compilation. This chapter also introduces pragmas, which are instructions to the compiler that are placed in the source file.

Appendix A, "Differences," lists the differences between Microsoft C and the description of the C language found in Appendix A of *The C Programming Language* by Brian W. Kernighan and Dennis M. Ritchie.

Appendix B, "Syntax Summary," summarizes the syntax of the C language as implemented by Microsoft.

The remainder of this chapter describes the notational conventions used throughout the manual.

1.3 Notational Conventions

This manual uses the following notational conventions:

Convention	Meaning
Bold	Bold type indicates text that must be typed exactly as shown. Text that is shown in bold type includes C keywords, such as **goto** and **char**, and operators, such as the addition operator (+) and the multiplication operator (*).
Italics	Italicized terms mark the places in syntax specifications and in the text where specific terms appear in an actual C program. For example, in **goto** *name*; *name* is italicized to show that this is a general form for the **goto** statement. In an actual program statement, the user supplies a particular identifier for the placeholder *name*. Occasionally, italics are used to emphasize particular words in the text.

`Examples`	Examples of C programs and program elements appear in a special typeface to look similar to listings on the screen or the output of commonly used computer printers:
	
	`int x, y;`
	`.`
	`.`
	`.`
	`swap (&x, &y);`

Ellipsis dots

Ellipsis dots may be vertical or horizontal. In the following example, the vertical ellipsis dots indicate that zero or more declarations, followed by one or more statements, may appear between the braces:

{
[[*declaration*]]
.
.
.
statement
[[*statement*]]
.
.
.
}

Vertical ellipsis dots are also used in program examples to indicate that a portion of the program has been omitted. For instance, in the following excerpt, two program lines are shown. The ellipsis dots between the lines indicate that intervening program lines occur but are not shown:

```
int x, y;
.
.
.
swap (&x, &y);
```

Horizontal ellipsis dots following an item indicate that more items having the same form may appear. For instance,

= { *expression* [[, *expression*]]...}

indicates that one or more expressions separated by commas may appear between the braces ({ }).

[[Double brackets]]

Double brackets enclose optional items in syntax specifications. For example,

return [[*expression*]];

is a syntax specification showing that *expression* is an optional item in the **return** statement.

"Quotation marks"	Quotation marks set off terms defined in the text. For example, the term "token" appears in quotation marks when it is defined. Some C constructs, such as strings, require quotation marks. Quotation marks required by the language have the form "" rather than " ". For example, `"abc"` is a C string.
SMALL CAPITALS	Names of special key combinations, such as CONTROL-Z, appear in small capital letters.

Chapter 2
Elements of C

2.1 Introduction 11
2.2 Character Sets 11
2.2.1 Letters and Digits 12
2.2.2 White-Space Characters 12
2.2.3 Punctuation and Special Characters 12
2.2.4 Escape Sequences 13
2.2.5 Operators 15
2.3 Constants 17
2.3.1 Integer Constants 17
2.3.2 Floating-Point Constants 19
2.3.3 Character Constants 20
2.3.4 String Literals 21
2.4 Identifiers 22
2.5 Keywords 24
2.6 Comments 24
2.7 Tokens 25

2.1 Introduction

This chapter describes the elements of the C programming language. The elements of the language are the names, numbers, and characters used to construct a C program. In particular, this chapter describes the following:

- Character sets
- Constants
- Identifiers
- Keywords
- Comments
- Tokens

2.2 Character Sets

Two character sets are defined for use in C programs: the C character set and the representable character set. The C character set consists of the letters, digits, and punctuation marks that have a specific meaning to the C compiler. C programs are constructed by combining the characters of the C character set into meaningful statements.

The C character set is a subset of the representable character set. The representable character set consists of all letters, digits, and symbols that a user can represent graphically with a single character. The extent of the representable character set depends on the type of terminal, console, or character device being used.

A C program can contain only characters from the C character set, with the exceptions of string literals, character constants, and comments, which can use any representable character. Each character in the C character set has an explicit meaning to the C compiler. The compiler generates error messages when it encounters misused characters or characters not belonging to the C character set.

The following sections describe the characters and symbols of the C character set and explain how and when to use them.

2.2.1 Letters and Digits

The C character set includes the uppercase and lowercase letters of the English alphabet and the 10 decimal digits of the Arabic number system:

- Uppercase English letters

  A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
- Lowercase English letters

  a b c d e f g h i j k l m n o p q r s t u v w x y z
- Decimal digits

  0 1 2 3 4 5 6 7 8 9

These letters and digits can be used to form the constants, identifiers, and keywords described later in this chapter.

The C compiler treats uppercase and lowercase letters as distinct characters. If a lowercase a is specified in a given item, you cannot substitute an uppercase A in its place; you must use the lowercase letter.

2.2.2 White-Space Characters

Space, tab, line-feed, carriage-return, form-feed, vertical-tab, and new-line characters are called white-space characters because they serve the same purpose as the spaces between words and lines on a printed page. These characters separate user-defined items, such as constants and identifiers, from other items within a program.

A CONTROL-Z character is treated as an end-of-file indicator. The compiler disregards any text following the CONTROL-Z mark.

The C compiler ignores white-space characters unless they are used as separators or as components of character constants or string literals. This means you can use extra white-space characters to make a program more readable. Comments are also treated as white space (see Section 2.6).

2.2.3 Punctuation and Special Characters

The punctuation and special characters in the C character set are used for a variety of purposes, from organizing the text of a program to defining the tasks to be carried out by the compiler or by the compiled program. Table 2.1 lists these characters.

Table 2.1
Punctuation and Special Characters

Character	Name	Character	Name
,	Comma	!	Exclamation mark
.	Period	¦	Vertical bar
;	Semicolon	/	Forward slash
:	Colon	\	Backslash
?	Question mark	~	Tilde
'	Single quotation	_	Underscore
"	Double quotation	#	Number sign
(	Left parenthesis	%	Percent sign
)	Right parenthesis	&	Ampersand
[	Left bracket	^	Caret
]	Right bracket	*	Asterisk
{	Left brace	–	Minus sign
}	Right brace	=	Equal sign
<	Left angle bracket	+	Plus sign
>	Right angle bracket		

These characters have special meaning to the C compiler; their use in the C language is described throughout this manual. Punctuation characters in the representable character set that do not appear in this list can be used only in string literals, character constants, and comments.

2.2.4 Escape Sequences

Escape sequences are special character combinations that represent white-space and nongraphic characters in strings and character constants. They are typically used to specify actions such as carriage returns and tab movements on terminals and printers and to provide literal representations of characters that normally have special meanings, such as the double quote (") character. An escape sequence consists of a backslash followed by a letter or combination of digits. Table 2.2 lists the C language escape sequences.

Table 2.2
Escape Sequences

Escape Sequence	Name
\ **n**	New line
\ **t**	Horizontal tab
\ **v**	Vertical tab
\ **b**	Backspace
\ **r**	Carriage return
\ **f**	Form feed
\ **a**	Bell (alert)
\ **'**	Single quote
\ **"**	Double quote
\ \	Backslash
\ *ddd*	ASCII character in octal notation
\ **x***dd*	ASCII character in hexadecimal notation

If the backslash precedes a character not included in the list above, the backslash is ignored and that character is represented literally. For example the pattern \c represents the character c in a string literal or character constant.

The sequences \ *ddd* and \ **x***dd* allow any character in the ASCII (American Standard Code for Information Interchange) character set to be given as a three-digit octal or a two-digit hexadecimal character code. For example, the backspace character can be given as \010 or \x08. The ASCII null character can be given as \0 or \x0.

Only the digits 0 through 7 can appear in an octal escape sequence, and at least one digit must appear. However, fewer than three digits can be specified. For example, the backspace character can be given in octal notation as \10. Similarly, a hexadecimal escape sequence must contain at least one digit, but the second digit can be omitted. Thus, the hexadecimal escape sequence for the backspace character can be given either as \x08 or as \x8.

Note

When using octal and hexadecimal escape sequences in strings, it is safest to give all digits of the escape sequence (three digits for octal and two digits for hexadecimal escape sequences). Otherwise, if the character immediately following the escape sequence happens to be an octal or hexadecimal digit, it is interpreted as part of the sequence. For example, if the string `\x7Bell` were printed, the result would be `{ell` because `\x7B` is interpreted as the ASCII left brace character (`{`). The string `\x07Bell` (note the `0`) is the correct way to represent the bell character followed by the word `Bell`.

Escape sequences allow nongraphic control characters to be sent to a display device. For example, the escape character, `\033`, is often used as the first character of a control command for a terminal or printer.

Nongraphic characters should always be represented by escape sequences because using a nongraphic character in a C program has unpredictable results.

The backslash character (\) used to introduce escape sequences also functions as a continuation character in strings and in preprocessor definitions. When a new-line character follows the backslash, the new line is disregarded, and the next line is treated as part of the previous line.

2.2.5 Operators

Operators are special character combinations that specify how values are to be transformed and assigned. The compiler interprets each of these character combinations as a single unit, called a "token" (see Section 2.7).

Table 2.3 lists the characters that form C operators and gives the name of each operator. Operators must be specified exactly as they appear in the tables, with no white space between the characters of multicharacter operators. The **sizeof** operator is not included in this table; it consists of a keyword (**sizeof**) rather than a symbol.

Table 2.3
Operators

Operator	Name
!	Logical NOT
~	Bitwise complement
+	Addition
–	Subtraction, arithmetic negation
*	Multiplication, indirection
/	Division
%	Remainder
<<	Left shift
>>	Right shift
<	Less than
<=	Less than or equal
>	Greater than
>=	Greater than or equal
==	Equality
!=	Inequality
&	Bitwise AND, address of
|	Bitwise inclusive OR
^	Bitwise exclusive OR
&&	Logical AND
||	Logical OR
,	Sequential evaluation
?:	Conditional[a]
++	Increment
--	Decrement
=	Simple assignment
+=	Addition assignment
-=	Subtraction assignment
*=	Multiplication assignment
/=	Division assignment
%=	Remainder assignment
>>=	Right-shift assignment

Table 2.3 *(continued)*

Operator	Name
<<=	Left-shift assignment
&=	Bitwise AND assignment
|=	Bitwise inclusive OR assignment
^=	Bitwise exclusive OR assignment

[a] The conditional operator is a ternary operator, not a multicharacter operator. The form of a conditional expression is the following:
expression **?** *expression* **:** *expression*

For a complete description of each operator, see Chapter 5, "Expressions and Assignments."

2.3 Constants

A constant is a number, a character, or a string of characters that can be used as a value in a program. The value of a constant does not change from execution to execution.

The C language has four kinds of constants: integer constants, floating-point constants, character constants, and string literals. The following sections define the format and use of each.

2.3.1 Integer Constants

An integer constant is a decimal, octal, or hexadecimal number that represents an integer value. A decimal constant has the form

digits

where *digits* is one or more decimal digits (0 through 9).

An octal constant has the form

0*odigits*

where *odigits* is one or more octal digits (0 through 7). The leading 0 is required.

A hexadecimal constant has one of the following forms:

0x*hdigits*
0X*hdigits*

where *hdigits* is one or more hexadecimal digits (0 through 9 and either uppercase or lowercase "a" through "f"). The leading 0 is required and must be followed by **x** or **X.**

No white-space characters can appear between the digits of an integer constant. Table 2.4 illustrates the form of integer constants.

Table 2.4

Examples of Integer Constants

Decimal Constants	Octal Constants	Hexadecimal Constants
10	012	0xa or 0xA
132	0204	0x84
32179	076663	0x7dB3 or 0x7DB3

Integer constants always specify positive values. If negative values are required, the minus sign (–) can be placed in front of the constant to form a constant expression with a negative value. The minus sign is treated as an arithmetic operator.

Every integer constant is given a type based on its value. A constant's type determines what conversions must be performed when the constant is used in an expression or when the minus sign (–) is applied. Decimal constants are considered signed quantities and are given **int** type, or **long** type if the size of the value requires it.

Octal and hexadecimal constants are given **int**, **unsigned int**, **long**, or **unsigned long** type depending on the size of the constant. If the constant can be represented as an **int**, it is given **int** type. If it is larger than the maximum positive value that can be represented by an **int**, but small enough to be represented in the same number of bits as an **int**, it is given **unsigned int** type. Similarly, a constant that is too large to be represented as an **unsigned int** is given **long** type, or **unsigned long** type, if necessary.

Table 2.5 shows the ranges of values and the corresponding types for octal and hexadecimal constants on a machine where the **int** type is 16 bits long.

Table 2.5

Types Assigned to Octal and Hexadecimal Constants

Hexadecimal Range	Octal Range	Type
0x0 – 0x7FFF	0 – 077777	int
0x8000 – 0xFFFF	0100000 – 0177777	unsigned int
0x10000 – 0x7FFFFFFF	0200000 – 017777777777	long
0x80000000 – 0xFFFFFFFF	020000000000 – 030000000000	unsigned long

The consequence of the above typing rules is that hexadecimal and octal constants are not sign-extended when they are converted to longer types. (For a discussion of type conversions, see Chapter 5, "Expressions and Assignments.")

The programmer can force any integer constant to be given **long** type by appending the letter "l" or "L" to the end of the constant. Table 2.6 illustrates **long** integer constants.

Table 2.6

Examples of Long Integer Constants

Decimal Constants	Octal Constants	Hexadecimal Constants
10L	012L	0xaL or 0xAL
791	01151	0x4f1 or 0x4F1

Types are described in Chapter 4, "Declarations," and conversions are described in Chapter 5, "Expressions and Assignments."

2.3.2 Floating-Point Constants

A floating-point constant is a decimal number representing a signed real number. The value of a signed real number includes an integer portion, a fractional portion, and an exponent. Floating-point constants have the form

[[*digits*]][[.*digits*]][[**E**[[–]]*digits*]]

where *digits* is one or more decimal digits (0 through 9), and **E** (or **e**) is the exponent symbol. Either the digits before the decimal point (the integer portion of the value) or the digits after the decimal point (the fractional portion) can be omitted, but not both. The decimal point can be omitted only when an exponent is given.

The exponent consists of the exponent symbol followed by a possibly negative constant integer value. No white-space characters can separate the digits or characters of the constant.

Floating-point constants always specify positive values. If negative values are required, the minus sign (–) can be placed in front of the constant to form a constant floating-point expression with a negative value. The minus sign is treated as an arithmetic operator.

The following examples illustrate some of the forms of floating-point constants and expressions:

```
15.75
1.575E1
1575e-2
-0.0025
-2.5e-3
25E-4
```

The integer portion of the floating-point constant can be omitted, as shown in the following examples:

```
.75
.0075e2
-.125
-.175E-2
```

All floating-point constants have type **double**.

2.3.3 Character Constants

A character constant is a letter, digit, punctuation character, or escape sequence enclosed in single quotation marks. The value of a character constant is the numerical representation of the character. Character constants consisting of more than one character or escape sequence are not allowed.

A character constant has the form

'*char*'

where *char* can be any character from the representable character set (including any escape sequence) except a single quotation mark ('), a backslash (\), or a new-line character. To use a single quotation mark or backslash character as a character constant, precede it with a backslash, as shown in Table 2.7. To represent a new-line character, use the escape sequence '**n**'.

Table 2.7
Examples of Character Constants

Constant	Value
'a'	Lowercase a
'?'	Question mark
'\b'	Backspace
'\x1B'	ASCII escape character
'\''	Single quotation mark
'\\'	Backslash

Character constants have type **int** and consequently are sign extended in type conversions (see Section 5.7 of Chapter 5, "Expressions and Assignments").

2.3.4 String Literals

A string literal is a sequence of letters, digits, and symbols enclosed in double quotation marks. A string literal is treated as an array of characters; each element of the array is a single character value.

The form of a string literal is

"*characters*"

where *characters* is zero or more characters from the representable character set, excluding the double quotation mark ("), the backslash (\), and the new-line character. To use the new-line character in a string, type a backslash immediately followed by a new-line character. The backslash

causes the new-line character to be ignored, allowing the programmer to form string literals that occupy more than one line. For example, the string literal

```
"Long strings can be bro\
ken into two pieces."
```

is identical to the string

```
"Long strings can be broken into two pieces."
```

To use the double quotation mark or backslash character within a string literal, precede it with a backslash, as shown in the following examples:

```
"This is a string literal."
"Enter a number between 1 and 100 \n or press Return"
"First\\Second"
"\"Yes, I do,\" she said."
"The following line shows a null string:"
""
```

Note that escape sequences (such as \\ and \") can appear in string literals. Each escape sequence counts as a single character.

The characters of a string are stored in order at contiguous memory locations. A null character (**\0**) is automatically appended to mark the end of the string. Each string in a program is considered to be a distinct item; if two identical strings appear in a program, they each receive distinct storage space.

String literals have type **char** [ ]. This means a string is an array whose elements have type **char**. The number of elements in the array is the number of characters in the string literal plus one, since the null character stored after the last character counts as an array element.

2.4 Identifiers

Identifiers are the names you supply for the variables, functions, and labels used in a given program. You create an identifier by declaring it with the associated variable or function. You can then use the identifier in later statements within the program to refer to the given item. (Declarations are described in Chapter 4, "Declarations.")

An identifier is a sequence of one or more letters, digits, or underscores (_) that begins with a letter or underscore. Any number of characters are allowed in a given identifier, but only the first 31 characters are significant to the compiler. (Other programs that read the compiler output, such as the linker, may use fewer characters.) Use leading underscores with care; identifiers beginning with an underscore can conflict with the names of hidden system routines and produce errors.

The following are examples of identifiers:

```
j
cnt
temp1
topofpage
skip12
```

The C compiler considers uppercase and lowercase letters to be separate and distinct characters. Therefore, you can create distinct identifiers that have the same spelling but different cases for one or more of the letters. For example, each of the following identifiers is unique:

```
add
ADD
Add
aDD
```

The C compiler does not allow an identifier that has the same spelling and case as a C language keyword. Keywords are described in Section 2.5.

Note

The linker may further restrict the number and type of characters for globally visible symbols, and, unlike the compiler, the linker may not distinguish between uppercase and lowercase letters. Consult your linker documentation for information on naming restrictions imposed by the linker.

2.5 Keywords

Keywords are predefined identifiers that have special meaning to the C compiler. They can be used only as defined. The names of program items cannot conflict with the keywords listed below:

auto	**double**	**int**	**struct**
break	**else**	**long**	**switch**
case	**enum**	**register**	**typedef**
char	**extern**	**return**	**union**
const	**float**	**short**	**unsigned**
continue	**for**	**signed**	**void**
default	**goto**	**sizeof**	**while**
do	**if**	**static**	**volatile**

Keywords cannot be redefined. However, you can specify text to be substituted for keywords before compilation by using C preprocessor directives (see Chapter 8, "Preprocessor Directives and Pragmas").

The **const** and **volatile** keywords are reserved for future use but have not yet been implemented.

The following identifiers may be keywords in some implementations. See your system documentation for more information.

cdecl
far
fortran
huge
near
pascal

2.6 Comments

A comment is a sequence of characters that is treated as a single white-space character by the compiler, but is otherwise ignored. A comment has the following form:

/* *characters* */

Here *characters* can be any combination of characters from the representable character set, including new-line characters but excluding the

combination */. This means that comments can occupy more than one line, but they cannot be nested.

Comments can appear anywhere a white-space character is allowed. The compiler ignores the characters of the comment, so keywords can appear in comments without producing errors. Since the compiler treats the comment as a single white-space character, comments cannot appear within a token.

The following examples illustrate some comments:

```
/* Comments can separate and document
   lines of a program. */

/* Comments can contain keywords such as for
   and while. */

/*************************************
   Comments can occupy several lines.
 *************************************/
```

Since comments cannot contain nested comments, the following example causes an error:

```
/* You cannot /* nest */ comments */
```

The compiler recognizes the first */, after the word `nest`, as the end of the comment. The compiler attempts to process the remaining text and produces an error when it cannot do so.

To suppress compilation of a large portion of a program or a program segment that contains comments, use the #**if** preprocessor directive instead of comments (see Section 8.4 of Chapter 8, "Preprocessor Directives and Pragmas").

2.7 Tokens

When the compiler processes a program, it breaks the program down into groups of characters known as "tokens." A token is a unit of program text that has meaning to the compiler and that cannot be broken down further. The operators, constants, identifiers, and keywords described in this chapter are examples of tokens. Punctuation characters such as brackets ([ ]), braces ({ }), angle brackets (< >), parentheses, and commas are also tokens.

Tokens are delimited by white-space characters and by other tokens, such as operators and punctuation symbols. To prevent the compiler from breaking an item down into two or more tokens, white-space characters are prohibited between the characters of identifiers, multicharacter operators, and keywords.

When the compiler interprets tokens, it incorporates as many characters as possible into a single token before moving on to the next token. Because of this behavior, tokens not separated by white space may not be interpreted as expected.

For example, consider the following expression:

```
i+++j
```

In the above example, the compiler first makes the longest possible operator (++) from the three plus signs, and then processes the remaining plus sign as an addition operator (+). This expression is interpreted as `(i++) + (j)`, not `(i) + (++j)`. Use white space and parentheses to clarify your intent in such cases.

Chapter 3
Program Structure

3.1 Introduction 29
3.2 Source Program 29
3.3 Source Files 30
3.4 Program Execution 32
3.5 Lifetime and Visibility 33
3.6 Naming Classes 36

3.1 Introduction

This chapter describes the structure of C language source programs and defines terms used later in this manual to describe the C language. It provides an overview of C language features that are described in detail in other chapters. The syntax and meaning of declarations and definitions are discussed in Chapter 4, "Declarations," and Chapter 7, "Functions." The C preprocessor and pragmas are described in Chapter 8, "Preprocessor Directives and Pragmas."

3.2 Source Program

A C source program is a collection of one or more directives, pragmas, declarations, and/or definitions. "Directives" instruct the C preprocessor to perform specific actions on the text of the program prior to compilation. "Pragmas" are instructions to the compiler that are carried out at compile time.

"Declarations" establish the names and attributes of variables, functions, and types used in the program. "Definitions" are declarations that also define variables and functions. A variable definition gives the initial value of the declared variable, in addition to its name and type. The definition causes storage to be allocated for the variable. A function definition specifies the function body, which is a compound statement containing the declarations and statements that constitute the function. The function definition also gives the function name, formal parameters, and return type.

A source program can have any number of directives, pragmas, declarations, and definitions. Each must have the appropriate syntax as described in this manual, and each can appear in any order in the program (subject to the rules outlined throughout this manual), although the order affects how variables and functions can be used in the program (see Section 3.5, "Lifetime and Visibility").

A nontrivial program always contains at least one definition, a function definition. The function defines the action to be taken by the program. The following example illustrates a simple C source program:

Example

```
int x = 1;                          /* Variable definitions */
int y = 2;

extern int printf(char *,...);/* Function declaration */

main ()                             /* Function definition
                                       for main function */
{
        int z;                      /* Variable declarations */
        int w;

        z = y + x;                  /* Executable statements */
        w = y - x;
        printf("z= %d \nw= %d \n", z, w);
}
```

This source program defines the function named `main` and declares the function `printf`. The variables `x` and `y` are defined with variable definitions; the variables `z` and `w` are just declared.

3.3 Source Files

Source programs can be divided into one or more source files. A C source file is a text file that contains all or part of a C source program; it may, for example, contain just a few of the functions needed by the program. When the source program is compiled, the individual source files that make up the program must be compiled individually and then linked. Separate source files can also be combined to form larger source files before compilation by using the #**include** directive, discussed in Chapter 8, "Preprocessor Directives and Pragmas."

A source file can contain any combination of complete directives, pragmas, declarations, and definitions. Items such as function definitions or large data structures cannot be split between source files.

A source file need not contain any executable statements. It is sometimes useful to place variable definitions in one source file and then declare references to these variables in other source files that use them. This makes the definitions easy to find and modify, if necessary. For the same reason, manifest constants and macros (discussed in Chapter 8, "Preprocessor Directives and Pragmas") are often organized into separate "include" files and inserted into source files where required.

Directives in a source file apply to that source file and its include files only. Moreover, each directive applies only to the portion of the file following the directive. If a common set of directives is to be applied to a source program, then all source files in the program must contain these directives.

Pragmas usually take effect over a specific region of a source file. However, the specific compiler action that is defined by a pragma is determined by the implementation. For a discussion of the effects of particular pragmas, see your system documentation.

The following is an example of a C source program contained in two source files. The `main` and `max` functions are assumed to be in separate files, and execution of the program is assumed to begin with the `main` function.

Example

```
/**************************************************************
        Source file 1 - main function
 **************************************************************/

#define ONE     1
#define TWO     2
#define THREE   3

extern int max(int, int);       /* Function declaration */

main ()                         /* Function definition */
{
        int w = ONE, x = TWO, y = THREE;
        int z = 0;
        z = max(x,y);
        w = max(z,w);
}

/**************************************************************
        Source file 2 - max function
 **************************************************************/

int max (a, b)                  /* Function definition */
int a, b;
{
        if ( a > b )
                return (a);
        else
                return (b);
}
```

In the first source file, the function `max` is declared without being defined. This is known as a "forward declaration." The function definition for `main` includes function calls to `max`.

The lines beginning with a number sign (#) are preprocessor directives. These directives instruct the preprocessor to replace the identifiers ONE, TWO, and THREE with the specified number in the first source file. The directives do not apply to the second source file.

The second source file contains the function definition for `max`. This definition satisfies the calls to `max` in the first source file. Once the source files are compiled, they can be linked and executed as a single program.

3.4 Program Execution

Every program has a primary (main) program function. In C, the primary program function must be named **main**. The **main** function serves as the starting point for program execution and usually controls execution of the program by directing the calls to other functions in the program. A program usually stops executing at the end of the **main** function, although it can stop at other points in the program, depending on the execution environment.

The source program usually has more than one function, each designed to perform one or more specific tasks. The **main** function can call these functions to perform the tasks. When a function is called, execution begins at the first statement in the called function. The function returns control when a **return** statement is executed or the end of the function is encountered.

All functions, including the **main** function, can be declared to have parameters. Functions called by other functions receive values for the parameters from the calling functions. Parameters of the **main** function can be declared to receive values passed to the **main** function from outside the program. For example, they can receive values from the command line when the program is executed.

When the **main** function takes parameters, C requires the first two parameters to be named **argc** and **argv**. The **argc** parameter is declared to hold the total number of arguments passed to the **main** function. The **argv** parameter is declared as an array of pointers, each element of which points to a string representation of an argument passed to the **main** function.

Traditionally, the third parameter to the **main** function (if there is a third parameter) is given the name **envp**. The C language does not require this name, however. The **envp** parameter is a pointer to a table of string values that set up the environment in which the program executes.

The operating system supplies values for the **argc**, **argv**, and **envp** parameters, and the user supplies the actual arguments to the **main** function. The argument-passing convention in use on a particular system is determined by the operating system rather than by the C language. For more information, see your system documentation.

Formal parameters to functions must be declared when the function is defined. Function definitions are described in more detail in Section 7.2 of Chapter 7, "Functions." Function declarations are discussed in Section 4.5 of Chapter 4, "Declarations."

3.5 Lifetime and Visibility

Two concepts, "lifetime" and "visibility," are important in understanding the structure of a C program. The lifetime of a variable or function can be either "global" or "local." An item with a global lifetime has storage and a defined value throughout the duration of the program; an item with a local lifetime is allocated new storage each time the "block" in which it is defined or declared is entered. When the block is exited, the local item loses its storage, and hence its value. Blocks are defined and discussed below.

An item is said to be "visible" in a block or source file if the type and name of the item are known in the block or source file. An item can also be "globally visible," which means that it is visible, or can be made visible through appropriate declarations, throughout all the source files that constitute the program. Visibility between source files (also known as "linkage") is discussed in greater detail in Section 4.6 of Chapter 4, "Declarations."

A block is a compound statement. Compound statements consist of declarations and statements, as described in Section 6.3 of Chapter 6, "Statements." The bodies of C functions are compound statements. Blocks can be nested; function bodies frequently contain blocks, which in turn can contain other blocks.

Declarations and definitions within blocks occur at the "internal level." Declarations and definitions outside all blocks occur at the "external level."

Both variables and functions can be *declared* at the external level or at the internal level. Variables can also be *defined* at the internal level, but functions can only be defined at the external level.

All functions have global lifetimes, regardless of where they are declared. Variables declared at the external level always have global lifetimes. Variables declared at the internal level usually have local lifetimes; however, the storage-class specifiers **static** and **extern** can be applied to declare global variables or references to global variables within a block. See Section 4.6 of Chapter 4, "Declarations," for a discussion of these options.

Variables declared or defined at the external level are visible from the point at which they are declared or defined to the end of the source file. These variables can be made visible in other source files with appropriate declarations, as described in Section 4.6, "Storage Classes." However, variables that are given **static** storage class at the external level are visible only within the source file in which they are defined.

In general, variables declared or defined at the internal level are visible from the point at which they are first declared or defined to the end of the block in which the definition or declaration appears. These variables are called local variables. If a variable declared inside a block has the same name as a variable declared at the external level, the block definition supersedes the external level definition of the variable for the duration of the block. The visibility of the external level variable is restored when the block is exited.

Block visibility can nest. This means that a block nested inside another block can contain declarations that redefine variables declared in the outer block. The redefinition of the variable holds in the inner block, but the original definition is restored when control returns to the outer block. Variables from outer blocks are visible inside all inner blocks, as long as they are not redefined in the inner blocks.

Functions with **static** storage class are visible only in the source file in which they are defined. All other functions are globally visible. For more information on function declarations, see Section 4.5 of Chapter 4, "Declarations."

Table 3.1 summarizes the main factors that determine the lifetime and visibility of functions and variables. The table is not, however, intended to cover all cases. Refer to the above discussion and to Section 4.6, "Storage Classes," for more information.

Table 3.1

Summary of Lifetime and Visibility

Level	Item	Storage Class Specifier	Lifetime	Visibility
External	Variable declaration	**static**	Global	Restricted to single source file
	Variable declaration	**extern**	Global	Remainder of source file
	Function declaration or definition	**static**	Global	Restricted to single source file
	Function declaration or definition	**extern**	Global	Remainder of source file
Internal	Variable definition or declaration	**extern** or **static**	Global	Block
	Variable definition or declaration	**auto** or **register**	Local	Block

The following program example illustrates blocks, nesting, and visibility of variables:

Example

```
/* i defined at external level */
int i = 1;

/* main function defined at external level */
main ()
{

    /* prints 1 (value of external level i) */
    printf("%d\n", i);

    /* first nested block */
    {

        /* i and j defined at internal level */
```

```
int i = 2, j = 3;

/* prints 2, 3 */
printf("%d\n%d\n", i, j);

/* second nested block */
{

    /* i is redefined */
    int i = 0;

    /* prints 0, 3 */
    printf("%d\n%d\n", i, j);

/* end of second nested block */
}

    /* prints 2 (outer definition restored) */
    printf("%d\n", i);

/* end of first nested block */
}

    /* prints 1 (external level definition restored) */
    printf("%d\n", i);
}
```

In this example, there are four levels of visibility: the external level and three block levels. Assuming that the function `printf` is defined elsewhere in the program, the `main` function prints out the values 1, 2, 3, 0, 3, 2, 1.

3.6 Naming Classes

In any C program, identifiers are used to refer to many different kinds of items. When you write a C program, you provide identifiers for the functions, variables, formal parameters, union members, and other items the program uses. C allows you to use the same identifier for more than one program item, as long as you follow the rules outlined in this section.

The compiler sets up "naming classes" to distinguish between the identifiers for different kinds of items. The names within each class must be unique to avoid conflict, but an identical name can appear in one or more naming classes. This means that you can use the same identifier for

two or more different items if the items are in different naming classes. The context of a given identifier in the program allows the compiler to resolve the reference without ambiguity.

The kinds of items you can name in C programs, and the rules for naming them, are described as follows:

Items	Naming Class
Variables and functions	The names of variables and functions are in a naming class with formal parameters and enumeration constants. Variable and function names must, therefore, be distinct from other names in this class with the same visibility. However, variable names can be redefined within program blocks, as described in Section 3.5, "Lifetime and Visibility." Function names can also be redefined in this manner.
Formal parameters	The names of formal parameters to a function are grouped with the names of the function's variables, so the formal parameter names should be distinct from the variable names. Redeclaring formal parameters within the function causes an error.
Enumeration constants	Enumeration constants are in the same naming class as variable and function names. This means that names of enumeration constants must be distinct from all variable and function names with the same visibility, and distinct from the names of other enumeration constants with the same visibility. However, like variable names, the names of enumeration constants have nested visibility, meaning that they can be redefined within blocks. See Section 3.5, "Lifetime and Visibility."
typedef names	The names of types defined with **typedef** are in a naming class with variable and function names. They must, therefore, be distinct from all variable and function

	names with the same visibility, and also from the names of formal parameters and enumeration constants. Like variable names, names used for **typedef** types can be redefined within program blocks. See Section 3.5, "Lifetime and Visibility."
Tags	Enumeration, structure, and union tags are grouped together in a single naming class. Each enumeration, structure, or union tag must be distinct from other tags with the same visibility. Tags do not conflict with any other names.
Members	The members of each structure and union form a naming class. The name of a member must, therefore, be unique within the structure or union, but it does not have to be distinct from any other name in the program, including names of members of different structures and unions.
Statement labels	Statement labels form a separate naming class. Each statement label must be distinct from all other statement labels in the same function. Statement labels do not have to be distinct from any other names or from label names in other functions.

Example

```
struct student {
        char student[20];
        int class;
        int id;
        } student;
```

Structure tags, structure members, and variable names are in three different naming classes, so no conflict occurs among the three items named `student` in the above example. The compiler determines how to interpret each occurrence of `student` by its context in the program. For example, when `student` appears after the **struct** keyword, it is known to be a structure tag. When `student` appears after a member-selection operator (–> or .), the name refers to the structure member. In other contexts, the identifier `student` refers to the structure variable.

Chapter 4
Declarations

4.1 Introduction 41
4.2 Type Specifiers 42
4.3 Declarators 46
4.3.1 Pointer, Array, and Function Declarators 47
4.3.2 Complex Declarators 48
4.3.3 Declarators with Special Keywords 51
4.4 Variable Declarations 53
4.4.1 Simple Variable Declarations 54
4.4.2 Enumeration Declarations 55
4.4.3 Structure Declarations 57
4.4.4 Union Declarations 60
4.4.5 Array Declarations 62
4.4.6 Pointer Declarations 64
4.5 Function Declarations 65
4.6 Storage Classes 68
4.6.1 Variable Declarations at the External Level 69
4.6.2 Variable Declarations at the Internal Level 72
4.6.3 Function Declarations at the External and Internal Levels 74
4.7 Initialization 75
4.7.1 Fundamental and Pointer Types 76
4.7.2 Aggregate Types 77
4.7.3 String Initializers 80
4.8 Type Declarations 80

4.8.1 Structure, Union, and Enumeration Types 81
4.8.2 Typedef Declarations 82
4.9 Type Names 83

4.1 Introduction

This chapter describes the form and constituents of C declarations for variables, functions, and types. C declarations have the form

[[*sc-specifier*]] [[*type-specifier*]] *declarator*[[=*initializer*]] [[,*declarator*[[=*initializer*]]...]]

where *sc-specifier* is a storage-class specifier, *type-specifier* is the name of a defined type, *declarator* is an identifier that can be modified to declare a pointer, array, or function, and *initializer* gives a value or sequence of values to be assigned to the variable being declared.

All C variables must be explicitly declared before they are used. C functions can be declared explicitly in a function declaration or implicitly by calling the function before it is declared or defined.

The C language defines a standard set of data types. You can add to that set by declaring new data types based on types already defined. You can declare arrays, data structures, and pointers to both variables and functions.

C declarations require one or more *declarators.* A declarator is an identifier that can be modified with brackets ([ ]), asterisks (*), or parentheses to declare an array, pointer, or function type. When you declare simple variables (such as character, integer, and floating-point values), or structures and unions of simple variables, the declarator is just an identifier.

Four storage-class specifiers are defined in C: **auto**, **extern**, **register**, and **static**. The storage-class specifier of a declaration affects how the declared item is stored and initialized and which portions of a program can reference it. The location of the declaration within the source program and the presence or absence of other declarations of the variable are also important factors in determining the visibility of variables.

Function declarations are presented in Section 4.5. For information on function definitions, see Section 7.2.

4.2 Type Specifiers

The C language provides definitions for a set of basic data types, called "fundamental" types. Their names are listed in Table 4.1.

Table 4.1
Fundamental Types

Integral Types[a]	Floating-Point Types[a]	Other
signed char	**float**	**void**[b]
signed int	**double** (also called long float)	
signed short int		
signed long int		
unsigned char		
unsigned int		
unsigned short int		
unsigned long int		

[a] Used to declare variables and function return types

[b] Used only to declare function return types

Enumeration types are also considered fundamental types. Type specifiers for enumeration types are discussed in Section 4.8.1. The **signed char**, **signed int**, **signed short int**, and **signed long int** types, together with their **unsigned** counterparts, are called "integral" types. The **float** and **double** type specifiers refer to "floating-point" types. Variable and function declarations can use any of the integral or floating-point type specifiers.

The **void** type can be used only to declare functions that return no value. Function types are discussed in Section 4.5, "Function Declarations."

You can create additional type specifiers with **typedef** declarations, discussed in Section 4.8.2.

Type specifiers are commonly abbreviated, as shown in Table 4.2. Integral types are signed by default. Thus, if the **unsigned** keyword is omitted from the type specifier, the integral type is signed, even if the **signed** keyword is not specified.

In some implementations, a compiler option may be available to change the default for the **char** type from signed to unsigned type. When such an option is in effect, the abbreviation **char** has the same meaning as **unsigned char**, and the **signed** keyword must be used to declare a signed character value.

Table 4.2

Type Specifiers and Abbreviations

Type Specifier	Abbreviations
signed char[a]	char
signed int	signed, int
signed short int	short, signed short
signed long int	long, signed long
unsigned char[b]	--
unsigned int	unsigned
unsigned short int	unsigned short
unsigned long int	unsigned long
float	--
long float	double

[a] When the **char** type is made unsigned by default (through the use of a compiler option), **signed char** cannot be abbreviated.

[b] When the **char** type is made unsigned by default (through the use of a compiler option), **unsigned char** can be abbreviated as **char**.

Note

This manual generally uses the abbreviated forms listed in Table 4.2 rather than the long forms of the type specifiers and assumes that the **char** type is signed by default. Therefore, throughout this manual, **char** stands for **signed char**.

Table 4.3 summarizes the storage associated with each fundamental type and gives the range of values that can be stored in a variable of each type. Since the **void** type does not apply to variables, it is not included in the table.

Table 4.3

Storage and Range of Values for Fundamental Types

Type	Storage	Range of Values (Internal)
char	1 byte	– 128 to 127
int	implementation dependent	
short	2 bytes	– 32768 to 32767
long	4 bytes	– 2,147,483,648 to 2,147,483,647
unsigned char	1 byte	0 to 255
unsigned	implementation dependent	
unsigned short	2 bytes	0 to 65535
unsigned long	4 bytes	0 to 4,294,967,295
float	4 bytes	IEEE standard notation; discussed below
double	8 bytes	IEEE standard notation; discussed below

The **char** type is used to store a letter, digit, or symbol from the representable character set. The integer value of a character is the ASCII code corresponding to that character. Since the **char** type is interpreted as a signed 1-byte integer, values in the range –128 to 127 are permitted for

char variables, although only the values from 0 to 127 have character equivalents. Similarly, the **unsigned char** type can store values in the range 0 to 255.

Note that the storage and range associated with the **int** and **unsigned int** types are not defined by the C language. Instead, the size of an **int** (signed or unsigned) corresponds to the natural size of an integer on a given machine. For example, on a 16-bit machine the **int** type is usually 16 bits, or 2 bytes. On a 32-bit machine the **int** type is usually 32 bits, or 4 bytes. Thus, the **int** type is equivalent either to the **short int** or the **long int** type, depending on the implementation. Similarly, the **unsigned int** type is equivalent either to the **unsigned short** or **unsigned long** type.

The **int** and **unsigned int** type specifiers are widely used in C programs because they allow a particular machine to handle integer values in the most efficient way for that machine. However, since the size of the **int** and **unsigned int** types varies, programs that depend on a specific **int** size may be nonportable. Expressions involving the **sizeof** operator (discussed in Section 5.3.4) can be used in place of hard-coded data sizes to increase the portability of the code.

The type specifiers **int** and **unsigned int** (or simply **unsigned**) are used to define certain features of the C language (for instance, for defining the **enum** type later in Section 4.8.1). In these cases, the definition of **int** and **unsigned int** for a particular implementation determines the actual storage.

Range of Values

The range of values for a variable lists the minimum and maximum values that can be represented *internally* in a given number of bits. However, because of C's conversion rules (discussed in detail in Chapter 5, "Expressions and Assignments"), it is not always possible to use the maximum or minimum for a constant of a given type in an expression.

For example, the constant expression –32768 consists of the arithmetic negation operator (–) applied to the constant value 32768. Since 32768 is too large to represent as a **short**, it is given **long** type, and the constant expression –32768 consequently has **long** type. The value –32768 can only be represented as a **short** by type casting it to the **short** type. No information is lost in the type cast, since –32768 can be represented internally in 2 bytes of storage space.

Similarly, a value such as 65000 can only be represented as an **unsigned short** by type casting the value to **unsigned short** type or by giving the value in octal or hexadecimal notation. The value 65000 in decimal notation is considered a signed constant, and is given **long** type because 65000 does not fit into a **short.** This **long** value can then be cast to the **unsigned short** type without loss of information, since 65000 will fit into 2 bytes of storage space when it is stored as an unsigned number.

Octal and hexadecimal constants may have either signed or unsigned type, depending on their size (see Section 2.3.1 for more information). However, the method used for assigning types to these constants ensures that they always behave like unsigned integers in type conversions.

Floating-point numbers use the IEEE (Institute of Electrical and Electronics Engineers, Inc.) format. Values with **float** type have 4 bytes, consisting of a sign bit, a 7-bit excess 127 binary exponent, and a 24-bit mantissa. The mantissa represents a number between 1.0 and 2.0. Since the high-order bit of the mantissa is always 1, it is not stored in the number. This representation gives a range of approximately 3.4E–38 to 3.4E+38.

Values with **double** type have 8 bytes. The format is similar to the **float** format, except that the exponent is 11 bits excess 1023, and the mantissa has 52 bits, plus the implied high-order 1 bit. This gives a range of approximately 1.7E–308 to 1.7E+308.

4.3 Declarators

Syntax

identifier
declarator[ ]
declarator[*constant-expression*]
**declarator*
declarator( )
declarator(*arg-type-list*)
(*declarator*)

C allows the programmer to declare *arrays* of values, *pointers* to values, and *functions returning* values of specified types. To declare these items, you must use a *declarator.*

A declarator is an identifier, possibly modified with brackets ([ ]), parentheses, and asterisks (*), to declare an array, pointer, or function type. Declarators appear in the pointer, array, and function declarations described in later sections of this chapter (Sections 4.4.6, 4.4.5, and 4.5, respectively). This section discusses the rules for forming and interpreting declarators.

4.3.1 Pointer, Array, and Function Declarators

When a declarator consists of an unmodified identifier, the item being declared has an unmodified type. Asterisks (*) can appear to the left of an identifier, modifying it to a *pointer* type. If the identifier is followed by brackets ([ ]), the type is modified to an *array* type. If the identifier is followed by parentheses, the type is modified to a *function returning* type.

A declarator does not constitute a complete declaration; a type specifier must be included as well. The type specifier gives the type of the elements for an array type, the type of object addressed by a pointer type, and the return type of a function.

The sections on pointer, array, and function declarations later in this chapter discuss each type of declaration in detail (see Sections 4.4.6, 4.4.5, and 4.5, respectively). The following examples illustrate the simplest forms of declarators:

Examples

```
int list[20];              /* Example 1 */

char *cp;                  /* Example 2 */

double func(void);         /* Example 3 */
```

The above examples declare the following:

1. An array of **int** values (`list`)
2. A pointer to a **char** value (`cp`)
3. A function with no arguments returning a **double** value (`func`)

4.3.2 Complex Declarators

Any declarator can be enclosed in parentheses. Parentheses are typically used to specify a particular interpretation of a "complex" declarator, as discussed below. A "complex" declarator is an identifier qualified by more than one array, pointer, or function modifier.

Various combinations of the array, pointer, and function modifiers can be applied to a single identifier. Some combinations are illegal. An array cannot be composed of functions, and a function cannot return an array or a function.

In interpreting complex declarators, brackets and parentheses (to the right of the identifier) take precedence over asterisks (to the left of the identifier). Brackets and parentheses have the same precedence and associate left to right. The type specifier is applied as the last step, when the declarator has been fully interpreted. Parentheses can be used to override the default association order in a way that forces a particular interpretation.

A simple rule that can be helpful in interpreting complex declarators is to read them "from the inside out." Start with the identifier and look to the right for brackets or parentheses. Interpret these (if any), then look to the left for asterisks. If you encounter a right parenthesis at any stage, go back and apply these rules to everything within the parentheses before proceeding. As the last step, apply the type specifier. To illustrate this rule, the steps are numbered in order in the following example:

```
char *(*(*var)())[10];
 ^    ^ ^ ^ ^  ^  ^
 7    6 4 2 1  3  5
```

1. The identifier `var` is declared as
2. a pointer to
3. a function returning
4. a pointer to
5. an array of 10 elements, which are
6. pointers to
7. **char** values.

The following examples provide further illustration and show how parentheses can affect the meaning of a declaration:

Examples

```
/********************* Example 1 *********************/

        /* array of pointers to int values */
int *var[5];

/********************* Example 2 *********************/

       /* pointer to array of int values */
int (*var)[5];

/********************* Example 3 *********************/

      /* function returning pointer to long */
long *var(long,long);

/********************* Example 4 *********************/

      /* pointer to function returning long */
long (*var)(long,long);

/********************* Example 5 *********************/

      /* array of pointers to functions
         returning structures */
struct both {
        int a;
        char b;
        } ( *var[5] )( struct both, struct both );

/********************* Example 6 *********************/

      /* function returning pointer
         to an array of 3 double values */
double ( *var( double (*)[3] ) )[3];

/********************* Example 7 *********************/

      /* array of arrays of pointers
         to pointers to unions */
union sign {
        int x;
        unsigned y;
        } **var[5][5];

/********************* Example 8 *********************/

      /* array of pointers to arrays
         of pointers to unions */
union sign *(*var[5])[5];
```

Example	Description
1	In the first example, the array modifier has higher priority than the pointer modifier, so `var` is declared to be an array. The pointer modifier applies to the type of the array elements; the elements are pointers to **int** values.
2	In the second example, parentheses alter the meaning of the declaration in the first example. Now the pointer modifier has higher priority than the array modifier, and `var` is declared to be a pointer to an array of five **int** values.
3	Function modifiers also have higher priority than pointer modifiers, so the third example declares `var` to be a function returning a pointer to a **long** value. The function is declared to take two **long** values as arguments.
4	The fourth example is similar to the second example. Parentheses give the pointer modifier higher priority than the function modifier, and `var` is declared to be a pointer to a function returning a **long** value. Again, the function takes two **long** arguments.
5	The elements of an array cannot be functions, but the fifth example demonstrates how to declare an array of pointers to functions instead. In this example, `var` is declared to be an array of five pointers to functions returning structures with two members. The arguments to the functions are declared to be two structures with the same structure type, `both`. Note that the parentheses surrounding `*var[5]` are required. Without them, the declaration is an illegal attempt to declare an array of functions, as shown below:
6	The sixth example shows how to declare a function returning a pointer to an array, since functions returning arrays are illegal. Here `var` is declared to be a function returning a pointer to an array of three **double** values. The function `var` takes one argument; the argument, like the return value, is a pointer to an array of three **double** values. The argument type is given by a complex abstract declarator. The parentheses around the asterisk in the argument type are required; without them, the argument type would be an array of three

```
                    /* ILLEGAL */
struct both *var[5]( struct both, struct both );
```

pointers to **double** values. For a discussion and examples of abstract declarators, see Section 4.9, "Type Names."

7 A pointer can point to another pointer, and an array can contain array elements, as the seventh example shows. Here `var` is an array of five elements. Each element is a five-element array of pointers to pointers to unions with two members.

8 The eighth example shows how the placement of parentheses alters the meaning of the declaration. In this example, `var` is a five-element array of pointers to five-element arrays of pointers to unions.

4.3.3 Declarators with Special Keywords

Your implementation of Microsoft C may include the following special keywords:

cdecl
far
fortran
huge
near
pascal

These keywords are used to modify the meaning of variable and function declarations. See your system documentation for a full discussion of the effects of these special keywords.

When a special keyword occurs in a declarator, it modifies the item immediately to the right of the keyword. More than one special keyword can be applied to the same item. For example, a function identifier might be modified with both the **far** and **pascal** keywords. The order of the keywords in this case does not matter (that is, **far pascal** and **pascal far** have the same effect).

Two or more special keywords can be used in different parts of the declaration to modify the meaning of the declaration. For example, the following declaration contains two occurrences of the **far** keyword:

```
int far * pascal far func(void);
```

The function identifier `func` is modified with the **pascal** and **far** keywords. The return value of `func` is declared to be a **far** pointer to an **int** value.

As in any C declaration, parentheses can be used to override the default interpretation of the declaration. The rules governing complex declarators (discussed in the previous section) apply to declarations using the special keywords as well.

The following examples show the use of special keywords in declarations:

Examples

```
/********************* Example 1 *********************/

int huge database[65000];

/********************* Example 2 *********************/

char * far * x;

/********************* Example 3 *********************/

double near cdecl calc(double,double);
double cdecl near calc(double,double);

/********************* Example 4 *********************/

char far fortran initlist[INITSIZE];
char far *nextchar, far *prevchar, far *currentchar;

/********************* Example 5 *********************/

char far *(far *getint)(int far *);
 ^    ^       ^        ^     ^    ^
 6    5       2        1     3    4
```

Example 1 declares a **huge** array named `database` with 65000 **int** elements. The **huge** keyword modifies the array declarator.

In Example 2, the **far** keyword modifies the asterisk to its right, making `x` a **far** pointer to a pointer to **char**. The declaration could be expressed equivalently in the following manner:

```
char * (far *x);
```

Example 3 shows two equivalent declarations. Both declare `calc` as a function with the **near** and **cdecl** attributes.

Example 4 also shows two declarations: the first declares a **far fortran** array of characters named `initlist`, and the second declares three **far** pointers named `nextchar`, `prevchar`, and `currentchar`. The pointers might be used to store the addresses of characters in the `initlist` array. Note that the **far** keyword must be repeated before each declarator.

Example 5 shows a more complex declaration with several occurrences of the **far** keyword. The steps in interpreting this declaration are as follows:

1. The identifier `getint` is declared as a
2. **far** pointer to
3. a function taking
4. a single argument that is a **far** pointer to an **int** value
5. and returning a **far** pointer to a
6. **char** value.

Note that the **far** keyword always modifies the item immediately to its right.

4.4 Variable Declarations

This section describes the form and meaning of variable declarations. In particular, it explains how to declare the following:

Type of Variable	Description
Simple variables	Single-value variables with integral or floating-point type.
Enumeration variables	Simple variables with integral type that hold one value from a set of named integer constants.
Structures	Variables composed of a collection of values that may have different types.
Unions	Variables composed of several values of different types occupying the same storage space.

Arrays	Variables composed of a collection of elements with the same type.
Pointers	Variables that point to other variables. These variables contain variable locations (in the form of addresses) instead of values.

The variable declarations discussed in this section have the general form

[[*sc-specifier*]] *type-specifier declarator* [[, *declarator...*]]

where *type-specifier* gives the data type of the variable and *declarator* is the variable's name, possibly modified to declare an array or a pointer type. More than one variable can be defined in the declaration by giving multiple declarators, separated by commas.

The *sc-specifier* gives the storage class of the variable. In some contexts, variables can be initialized when they are declared. For information on storage classes and initialization, see Sections 4.6 and 4.7, respectively.

4.4.1 Simple Variable Declarations

Syntax

type-specifier identifier [[, *identifier...*]];

A declaration for a simple variable defines the variable's name and type; it can also define the variable's storage class, as described in Section 4.6. The variable's name is the *identifier* given in the declaration. The *type-specifier* gives the name of a defined data type, as described below.

You can define several variables in the same declaration by giving a list of identifiers separated by commas (,). Each identifier in the list names a variable. All variables defined in the declaration have the same type.

Examples

```
int x;                          /* Example 1 */

unsigned long reply, flag;      /* Example 2 */

double order;                   /* Example 3 */
```

The first example defines a simple variable `x`. This variable can hold any value in the set defined by the **int** type in a particular implementation.

The second example defines two variables, `reply` and `flag`. Both variables have **unsigned long** type and hold unsigned integer values.

The third example defines a variable `order` that has **double** type. Floating-point values can be assigned to this variable.

4.4.2 Enumeration Declarations

Syntax

enum [[*tag*]] {*enum-list*} *identifier* [[, *identifier*...]];
enum *tag identifier* [[, *identifier*...]];

An enumeration declaration gives the name of the enumeration variable and defines a set of named integer constants (the "enumeration set"). A variable declared to have enumeration type stores any one of the values of the enumeration set defined by that type. The integer constants of the enumeration set have **int** type; thus, the storage associated with an enumeration variable is the storage required for a single **int** value.

Enumeration declarations begin with the **enum** keyword and have two forms, as shown above. In the first form, the values and names of the enumeration set are specified in the *enum-list*, described in detail below. The optional *tag* is an identifier that names the enumeration type defined by the *enum-list*. The *identifier* names the enumeration variable. More than one enumeration variable can be defined in the declaration.

The second form uses an enumeration *tag* to refer to an enumeration type. The *enum-list* does not appear in this type of declaration because the enumeration type is defined elsewhere. An error is generated if the given *tag* does not refer to a defined enumeration type or if the named type is not currently visible.

An *enum-list* has the following form:

identifier [[= *constant-expression*]]
[[, *identifier* [[= *constant-expression*]]]]
.
.
.

Each *identifier* names a value of the enumeration set. By default, the first identifier is associated with the value 0, the next identifier is associated with the value 1, and so on through the last identifier appearing in the declaration. The name of an enumeration constant is equivalent to its value.

The phrase "= *constant-expression*" overrides the default sequence of values. An identifier followed by the phrase "= *constant-expression*" is associated with the value given by *constant-expression*. The *constant-expression* must have **int** type and can be negative. The next identifier in the list is associated with the value of "*constant-expression* + 1", unless it is explicitly given another value.

An enumeration set can contain duplicate constant values, but each identifier in an enumeration list must be unique; that is, it must be different from all other enumeration identifiers with the same visibility. For example, the value 0 could be given to two different identifiers, `null` and `zero`, in the same set. The identifiers in the list must also be distinct from other identifiers with the same visibility, including ordinary variable names and identifiers in other enumeration lists. Enumeration tags must be distinct from other enumeration, structure, and union tags with the same visibility.

Examples

```
/********************* Example 1 *********************/

enum day {
        saturday,
        sunday = 0,
        monday,
        tuesday,
        wednesday,
        thursday,
        friday
        } workday;

/********************* Example 2 *********************/

enum day today = wednesday;
```

The first example defines an enumeration type named `day` and declares a variable named `workday` with that enumeration type. The value 0 is associated with `saturday` by default. The identifier `sunday` is explicitly set to 0. The remaining identifiers are given the values 1 through 5 by default.

In the second example, a value from the set is assigned to the variable `today`. Note that the name of the enumeration constant is used to assign the value. Since the `day` enumeration type was previously declared, only the enumeration tag is necessary in this declaration.

4.4.3 Structure Declarations

Syntax

struct [[*tag*]] { *member-declaration-list*} *declarator* [[, *declarator...*]];
struct *tag declarator* [[, *declarator...*]];

A structure declaration defines the name of the structure variable and specifies a sequence of variable values (called "members" of the structure) that can have different types. A variable with structure type holds the entire sequence defined by that type.

Structure declarations begin with the **struct** keyword and have two forms, as shown above. In the first form, the types and names of the structure members are specified in the *member-declaration-list*, described in detail below. The optional *tag* is an identifier that names the structure type defined by the *member-declaration-list*.

Each *declarator* gives the name of a structure variable. The *declarator* may also modify the type of the variable to a pointer to the structure type, an array of structures, or a function returning a structure.

The second form uses a structure *tag* to refer to a structure type. The *member-declaration-list* does not appear in this type of declaration because the structure type is defined elsewhere. The structure type definition must be visible for a *tag* declaration to be used, and the definition must appear prior to the *tag* declaration, unless the *tag* is used to declare a pointer variable or a **typedef** structure type. These declarations can use a structure *tag* before the structure type is defined, as long as the structure definition is visible to the declaration.

A *member-declaration-list* is a list of one or more variable or bit-field declarations. Each variable declared in the *member-declaration-list* is defined as a member of the structure type. Variable declarations within member declaration lists have the same form as the variable declarations discussed in this chapter, except that the declarations do not contain storage-class specifiers or initializers. The structure members can have any variable type: fundamental, array, pointer, union, or structure.

A member cannot be declared to have the type of the structure in which it appears. However, a member can be declared as a pointer to the structure type in which it appears, allowing you to create linked lists of structures.

Bit Fields

A bit-field declaration has the following form:

type-specifier [[*identifier*]] **:** *constant-expression*;

The bit field consists of the number of bits specified by *constant-expression*. The *type-specifier* for a bit-field declaration must specify an unsigned integral type, and the *constant-expression* must be a non-negative integer value. Arrays of bit fields, pointers to bit fields, and functions returning bit fields are not allowed. The optional *identifier* names the bit field. An unnamed bit field whose width is specified as 0 has a special function: it guarantees that storage for the member following it in the declaration list begins on an **int** boundary.

The identifiers in the structure declaration list must be unique within that list. It is not necessary for the identifiers in the list to be distinct from ordinary variable names or from identifiers in other structure declaration lists. Structure tags must be distinct from other structure, union, and enumeration tags having the same visibility.

Storage

Structure members are stored sequentially in the same order in which they are declared: the first member has the lowest memory address and the last member the highest. The storage for each member begins on a memory boundary appropriate to its type. Therefore, unnamed blanks can occur between the members of a structure in memory.

Bit fields are not stored across boundaries of their declared type. For example, a bit field declared with **unsigned int** type is either packed into the space remaining in the previous **unsigned int** or it begins a new **unsigned int**.

Examples

```
/******************** Example 1 ********************/

struct {
            float x,y;
    } complex;

/******************** Example 2 ********************/

struct employee {
            char name[20];
            int id;
            long class;
    } temp;

/******************** Example 3 ********************/

struct employee student, faculty, staff;

/******************** Example 4 ********************/

struct sample {
            char c;
            float *pf;
            struct sample *next;
    } x;

/******************** Example 5 ********************/

struct {
            unsigned icon : 8;
            unsigned color : 4;
            unsigned underline : 1;
            unsigned blink : 1;
    } screen[25][80];
```

The first example defines a structure variable named `complex`. This structure has two members with **float** type, `x` and `y`. The structure type is not named.

The second example defines a structure variable named `temp`. The structure has three members: `name`, `id`, and `class`. The `name` member is a 20-element array and `id` and `class` are simple members with **int** and **long** type, respectively. The identifier `employee` is the structure tag.

The third example defines three structure variables: `student`, `faculty`, and `staff`. Each structure has the same list of three members. The members are declared to have the structure type `employee`, defined in the previous example.

The fourth example defines a structure variable named `x`. The first two members of the structure are a **char** variable and a pointer to a **float** value. The third member, `next`, is declared as a pointer to the structure type being defined (`sample`).

The fifth example defines a two-dimensional array of structures named `screen`. The array contains 2000 elements, and each element is an individual structure containing four bit-field members: `icon`, `color`, `underline`, and `blink`.

4.4.4 Union Declarations

Syntax

union [[*tag*]] { *member-declaration-list*} *declarator* [[, *declarator*...]];
union *tag declarator*[[, *declarator*...]];

A union declaration defines the name of the union variable and specifies a set of variable values, called "members" of the union, that can have different types. A variable with union type stores any single value defined by that type.

Union declarations have the same forms as structure declarations except that they begin with the **union** keyword instead of the **struct** keyword. The same rules govern structure and union declarations, except that bit-field members are not allowed in unions.

The storage associated with a union variable is the storage required for the longest member of the union. When a smaller member is used, the union variable may contain unused memory space. All members are stored in the same memory space and start at the same address. The stored value is overwritten each time a value is assigned to a different member.

Examples

```
/******************** Example 1 ********************/

union sign {
        int svar;
        unsigned uvar;
    } number;

/******************** Example 2 ********************/

union {
        char *a, b;
        float f[20];
    } jack;

/******************** Example 3 ********************/

union {
        struct {
                char icon;
                unsigned color : 4;
            } window1, window2, window3, window4;
    } screen[25][80];
```

The first example defines a union variable named `number` that has two members: `svar`, a signed integer, and `uvar`, an unsigned integer. This declaration allows the current value of `number` to be stored as either a signed or an unsigned value. The union type is named `sign`.

The second example defines a union variable named `jack`. The members of the union are, in order of their declaration, a pointer to a **char** value, a **char** value, and an array of **float** values. The storage allocated for `jack` is the storage required for the 20-element array `f`, since `f` is the longest member of the union. The union type is unnamed.

The third example defines a two-dimensional array of unions named `screen`. The array contains 2000 elements. Each element is an individual union with four members: `window1`, `window2`, `window3`, and `window4`, where each member is a structure. At any given time, each union element holds one of the four possible structure members. Thus, the `screen` variable is a composite of up to four different "windows."

4.4.5 Array Declarations

Syntax

type-specifier declarator [*constant-expression*];
type-specifier declarator [ ];

A declaration for an array defines the name of the array and the type of each element. It can also define the number of elements in the array. A variable with array type is considered a pointer to the type of the array elements, as described in Section 5.2.2, "Identifiers."

Array declarations have two forms, as shown above. The *declarator* gives the variable name, and may modify the variable's type. The brackets ([ ]) following the *declarator* modify the declarator to array type. The *constant-expression* inside the brackets defines the number of elements in the array. Each element has the type given by the *type-specifier*, which can specify any type except **void** and function types.

The second form omits the *constant-expression* in brackets. This form can be used only if the array is initialized, declared as a formal parameter, or declared as a reference to an array explicitly defined elsewhere in the program.

Arrays of arrays, or "multidimensional" arrays, are defined by giving a list of bracketed *constant-expressions* following the array declarator:

type-specifier declarator[*constant-expression*] [*constant-expression*] ...

Each *constant-expression* in brackets defines the number of elements in a given dimension: two-dimensional arrays have two bracketed expressions, three-dimensional arrays have three, and so on. When a multidimensional array is declared within a function, the first *constant-expression* can be omitted if the array is initialized, declared as a formal parameter, or declared as a reference to an array explicitly defined elsewhere in the program.

Arrays of pointers to various types can be defined by using complex declarators, as described earlier in Section 4.3.2.

The storage associated with an array type is the storage required for all of its elements. The elements of an array are stored in contiguous and increasing memory locations, from the first element to the last. No blanks occur between the elements of an array in storage.

Arrays are stored by row. For example, the following array consists of two rows with three columns each:

```
char A[2][3];
```

The three columns of the first row are stored first, followed by the three columns of the second row. This means that the last subscript varies most quickly.

To refer to an individual element of an array, use a subscript expression, discussed in Section 5.2.5.

Examples

```
/******************** Example 1 ********************/

int scores[10], game;

/******************** Example 2 ********************/

float matrix[10][15];

/******************** Example 3 ********************/

struct {
            float x,y;
            } complex[100];

/******************** Example 4 ********************/

char *name[20];
```

The first example defines an array variable named `scores` with 10 elements, each of which has **int** type. The variable named `game` is declared as a simple variable with **int** type.

The second example defines a two-dimensional array named `matrix`. The array has 150 elements, each having **float** type.

The third example defines an array of structures. This array has 100 elements; each element is a structure containing two members.

The fourth example defines an array of pointers. The array has 20 elements, each of which is a pointer to a **char** value.

4.4.6 Pointer Declarations

Syntax

type-specifier ∗*declarator*;

A pointer declaration defines the name of the pointer variable and the type of the object to which the variable points. The *declarator* defines the variable's name, and may modify its type. The *type-specifier* gives the type of the object, which can be any fundamental, structure, or union type.

Pointer variables can also point to functions, arrays, and other pointers. For information on declaring more complex pointer types, refer to Section 4.3.2, "Complex Declarators."

A pointer to a structure or union type can be declared before the structure or union type is defined, as long as the pointer is not used before the type is defined. Such declarations are allowed because the compiler does not need to know the size of the structure or union to allocate space for the pointer variable. The pointer is declared by using the structure or union tag (see the fourth example below).

A variable declared as a pointer holds a memory address. The amount of storage required for an address and the meaning of the address depend on the given implementation of the compiler. Pointers to different types are not guaranteed to have the same length.

In some implementations, the special keywords **near**, **far**, and **huge** are available to modify the size of a pointer. Declarations using special keywords are described in Section 4.3.3. See your system documentation for more information on the meaning and use of these keywords.

Examples

```
char *message;                          /* Example 1 */

int *pointers[10]                       /* Example 2 */

int (*pointer)[10];                     /* Example 3 */

struct list *next, *previous;           /* Example 4 */

struct list {                           /* Example 5 */
            char *token;
            int count;
```

```
            struct list *next;
    } line;

struct id {                                    /* Example 6 */
        unsigned int id_no;
        struct name *pname;
    } record;
```

The first example defines a pointer variable named `message`. It points to a variable with **char** type.

The second example defines an array of pointers named `pointers`. The array has 10 elements; each element is a pointer to a variable with **int** type.

The third example defines a pointer variable named `pointer`; it points to an array with 10 elements. Each element in this array has **int** type.

The fourth example defines two pointer variables that point to the structure type `list`. This declaration can appear before the definition of the `list` structure type (see the next example), as long as the `list` type definition has the same visibility as the declaration.

The fifth example declares the variable `line` to have the structure type named `list`. The `list` structure type is defined to have three members; the first member is a pointer to a **char** value, the second is an **int** value, and the third is a pointer to another `list` structure.

The sixth example declares the variable `record` to have the structure type `id`. Note that `pname` is declared as a pointer to another structure type `name`. This declaration can occur before the `name` type is defined.

4.5 Function Declarations

Syntax

[[*type-specifier*]] *declarator*([[*arg-type-list*]]) [[, *declarator...*]];

A function declaration defines the name and return type of a function, and possibly establishes the types and number of arguments to the function. Function declarations, also called forward declarations, do not define the function body or parameters; instead, they permit the attributes of the function to be known before the function is defined. Function definitions are described in detail in Section 7.2.

The *declarator* of the function declaration names the function, and the *type-specifier* gives the function's return type. If the *type-specifier* is omitted from a function declaration, the return type of the function is assumed to be **int**.

Function declarations may include either the **extern** or the **static** storage-class specifier. Storage-class specifiers are discussed in Section 4.6.

Argument-Type List

The *arg-type-list* establishes the number and types of the arguments to the function, and has the following form:

type-name-list[[,...]]

The *type-name-list* is a list of one or more type names. Each *type-name* is separated from the next by a comma. The first *type-name* gives the type of the first argument to the function, the second *type-name* gives the type of the second argument, and so on. If the *arg-type-list* ends with a comma followed by three periods (,...), the number of arguments to the function is variable. However, the function is expected to have at least as many arguments as there are *type-names* preceding the last comma.

If the *arg-type-list* contains only three periods (...), the number of arguments to the function is variable and may be zero.

Note

To maintain compatibility with previous versions, the compiler will also accept the comma character, without the trailing periods, at the end of the *arg-type-list* to indicate a variable number of arguments. A single comma can also be used instead of three periods to form the *arg-type-list* of a function taking zero or more arguments. Use of the comma is supported only for compatibility; using the three periods is recommended for new code.

A *type-name* for a fundamental, structure, or union type consists of the type specifier for that type (such as **int**). The *type-names* for pointers, arrays, and functions are formed by combining a type specifier

with an "abstract declarator"; that is, a declarator without an identifier. Section 4.9, "Type Names," explains how to form and interpret abstract declarators.

The special keyword **void** can be used in place of the *arg-type-list* to declare a function that has no arguments. The compiler produces a warning message if a call to the function or a call to the function definition specifies arguments.

One other special construction is allowed in the *arg-type-list*. The phrase **void** * specifies an argument of any pointer type. This phrase can be used in the *arg-type-list* as if it were a *type-name*.

The *arg-type-list* may be omitted. In this case the parentheses after the function identifier are still required, but they are empty. In this form the function declaration establishes neither the number nor the types of arguments to the function. When this information is omitted, the compiler does not perform any type checking between the actual arguments in a function call and the formal parameters of the function definition. See Section 7.4, "Function Calls," for more information.

Return Type

Functions can return values of any type except arrays and functions. Therefore, the *type-specifier* of a function declaration can specify any fundamental, structure, or union type. The function identifier can be modified with one or more asterisks (*) to declare a pointer return type.

Although functions are not allowed to return arrays and functions, they can return pointers to arrays and functions. Functions that return pointers to array or function types are declared by modifying the function identifier with asterisks (*), brackets ([ ]), and parentheses to form a complex declarator. Forming and interpreting complex declarators is discussed in Section 4.3.2.

Examples

```
int add(int, int);                    /* Example 1 */

double calc();                        /* Example 2 */

char *strfind(char *,...);            /* Example 3 */

void draw(void);                      /* Example 4 */
```

```
double (*sum(double, double))[3];    /* Example 5 */

int (*select(void))(int);            /* Example 6 */

char *p;                             /* Example 7 */
short *q;
int prt(void *);
```

The first example declares a function named `add` that takes two **int** arguments and returns an **int** value.

The second example declares a function named `calc` that returns a **double** value. No argument-type list is given.

The third example declares a function named `strfind`, which returns a pointer to a **char** value. The function takes at least one argument, a pointer to a **char** value. The argument-type list ends with a comma followed by three periods, indicating that the function may take more arguments.

The fourth example declares a function with **void** return type (returning no value). The *argument-type-list* is also **void**, meaning no arguments are expected for this function.

In the fifth example, `sum` is declared as a function returning a pointer to an array of three **double** values. The `sum` function takes two arguments, each a **double** value.

In the sixth example, the function named `select` is declared to take no arguments and return a pointer to a function. The pointer return value points to a function taking one **int** argument and returning an **int** value.

In the seventh example, the function `prt` is declared to take a pointer argument of any type and return an **int**. Either the **char** pointer `p` or the **short** pointer `q` could be passed as an argument to `prt` without producing a type-mismatch warning.

4.6 Storage Classes

The storage class of a variable determines whether the item has a "global" or "local" lifetime. An item with a global lifetime exists and has a value throughout the duration of the program. All functions have global lifetimes.

Variables with local lifetimes are allocated new storage each time execution control passes to the block in which they are defined. When execution control passes out of the block, the variables no longer have meaningful values.

Although C defines only two types of storage classes, four storage-class specifiers are available. They are as follows:

auto
register
static
extern

Items with **auto** and **register** class have local lifetimes. The **static** and **extern** specifiers refer to items with global lifetimes.

The four storage-class specifiers have distinct meanings because storage-class specifiers affect the visibility of functions and variables, as well as their storage class. The term "visibility" refers to the portion of the source program in which the variable or function can be referenced. An item with a global lifetime exists throughout the execution of the source program, but it may not be "visible" in all parts of the program. Visibility and the related concept of lifetime are discussed in Section 3.5.

The placement of variable or function declarations within source files also affects storage class and visibility. Declarations outside all function definitions are said to occur at the "external level"; declarations within function definitions occur at the "internal level."

The exact meaning of each storage-class specifier depends on whether the declaration occurs at the external or internal level and whether the item declared is a variable or a function. The following sections describe the meaning of storage-class specifiers in each kind of declaration; they also explain the default behavior when the storage-class specifier is omitted from a variable or function declaration.

4.6.1 Variable Declarations at the External Level

Variable declarations at the external level use the **static** and **extern** storage-class specifiers or omit the storage-class specifier entirely. The **auto** and **register** storage-class specifiers are not allowed at the external level.

Variable declarations at the external level are either *definitions* of variables or *references* to variables defined elsewhere. An external variable declaration that also initializes the variable (implicitly or explicitly) is a definition of the variable. Definitions at the external level can take several forms:

- A variable can be defined at the external level by declaring it with the **static** storage-class specifier. The **static** variable can be explicitly initialized with a constant expression, as described in Section 4.7. If the initializer is omitted, the variable is automatically initialized to zero at compile time. Thus, `static int k = 16;` and `static int k;` are both considered definitions.
- A variable is defined when it is explicitly initialized at the external level. For example, `int j = 3;` is a variable definition.

Once a variable is defined at the external level, it is visible throughout the remainder of the source file in which it appears. The variable is not visible above its definition in the same source file, nor is it visible in other source files of the program, unless a *reference* is declared to make it visible, as described below.

A variable can be defined at the external level only once within a source file. If the **static** storage-class specifier is given, another variable with the same name can be defined with the **static** storage-class specifier in a different source file. Since each **static** definition is visible only in its own source file, no conflict occurs.

The **extern** storage-class specifier is used to declare a *reference* to a variable defined elsewhere. These declarations can be used to make a definition in another source file visible or to make a variable visible above its definition in the same source file. Once a reference to the variable is declared at the external level, the variable is visible throughout the remainder of the source file in which the declared reference occurs.

Declarations that use the **extern** storage-class specifier are not allowed to contain initializers, since they refer to variables whose values are already defined.

For an **extern** reference to be valid, the variable to which it refers must be defined once, and only once, at the external level. The definition can be in any of the source files that form the program.

One special case is not covered by the rules outlined above. You can omit both the storage-class specifier and the initializer from a variable declaration at the external level; for example, the declaration `int n;` is a valid external declaration. This declaration can have one of two different meanings, depending on the context:

1. If a variable by the same name is *defined* at the external level elsewhere in the program, the declaration is taken to be a reference to that variable, exactly as if the **extern** storage-class specifier had been used in the declaration.

2. If no such definition is present, the declared variable is allocated storage at link time and initialized to 0. If more than one such declaration appears in the program, storage is allocated for the largest size declared for the variable. For example, if a program contains two uninitialized declarations of `i` at the external level, `int i;` and `char i;`, storage space for an **int** is allocated for `i` at link time.

Uninitialized variable declarations at the external level are not recommended for any file that might be placed in a library.

Example

```
/*****************************************************
                    SOURCE FILE ONE
*****************************************************/

extern int i;                       /* reference to i,
                                       defined below */

main()
{
        i++;
        printf("%d\n", i);      /* i equals 4 */
        next();
}

int i = 3;                          /* definition of i */

next()
{
        i++;
        printf("%d\n", i);      /* i equals 5 */
        other();
}
```

```
/*************************************************************
                    SOURCE FILE TWO
*************************************************************/

extern int i;                       /* reference to i in
                                       first source file */

other()
{
        i++;
        printf("%d\n", i);    /* i equals 6 */
}
```

The two source files contain a total of three external declarations of `i`. Only one declaration contains an initialization; that declaration, `int i = 3;` , defines the global variable `i` with initial value 3. The **extern** declaration of `i` at the top of the first source file makes the global variable visible above its definition in the file. Without the **extern** declaration, the `main` function could not reference the global variable `i`. The **extern** declaration of `i` in the second source file makes the global variable visible in that source file.

All three functions perform the same task: they increase `i` and print it. (Assume that the `printf` function is defined elsewhere in the program.) The values printed are 4, 5, and 6.

If the variable `i` had not been initialized, it would have been automatically set to 0 at link time. The values printed in this case would be 1, 2, and 3.

4.6.2 Variable Declarations at the Internal Level

Any of the four storage-class specifiers can be used for variable declarations at the internal level. When the storage-class specifier is omitted from a variable declaration at the internal level, the default storage class is **auto**.

The **auto** storage-class specifier declares a variable with a local lifetime. The variable is visible only in the block in which it is declared. Declarations of **auto** variables can include initializers, as discussed later in this chapter. Variables with **auto** storage class are not initialized automatically, so they should be explicitly initialized when declared or assigned initial values in statements within the block. If not initialized, the values of **auto** variables are undefined.

The **register** storage-class specifier tells the compiler to give the variable storage in a register, if possible. Register storage usually results in faster access time and smaller code size. Variables declared with **register** storage class have the same visibility as **auto** variables.

The number of registers that can be used for variable storage is machine dependent. If no registers are available when the compiler encounters the **register** declaration, the variable is given **auto** storage class and stored in memory. The compiler assigns register storage to variables in exactly the same order in which the declarations appear in the source file. Register storage, if available, is only guaranteed for **int** and pointer types.

A variable declared at the internal level with the **static** storage-class specifier has a global lifetime, and is visible only within the block in which it is declared. Unlike **auto** variables, variables declared as **static** retain their values when the block is exited.

A **static** variable can be initialized with a constant expression. If not explicitly initialized, a **static** variable is automatically set to 0. Initialization is performed once, at compile time. The **static** variable is *not* reinitialized each time the block is entered.

A variable declared with the **extern** storage-class specifier is a reference to a variable with the same name defined at the external level in any of the source files of the program. The purpose of the internal **extern** declaration is to make the external-level variable definition visible within the block. The internal **extern** declaration does not change the visibility of the global variable in any other part of the program.

Example

```
int i = 1;

main()
{       /* reference to i, defined above */
        extern int i;

        /* initial value is zero; a is
           visible only within main */
        static int a;

        /* b is stored in a register, if possible */
        register int b = 0;
```

```
        /* default storage class is auto */
        int c = 0;

        /* values printed are 1, 0, 0, 0 */
        printf("%d\n%d\n%d\n%d\n", i, a, b, c);
        other();
}

other()
{
        /* i is redefined */
        int i = 16;

        /* this a is visible only within other */
        static int a = 2;

        a += 2;
        /* values printed are 16, 4 */
        printf("%d\n%d\n", i, a);
}
```

The variable `i` is defined at the external level with initial value 1. A reference to the external-level `i` is declared in the `main` function with an **extern** declaration. The **static** variable `a` is automatically set to 0, since the initializer is omitted. The call to `printf` (assuming the `printf` function is defined elsewhere in the source program) prints out the values 1, 0, 0, 0.

In the `other` function, the variable `i` is redefined as a local variable with initial value 16. This does not affect the value of the external-level `i`. The variable `a` is declared as a **static** variable and initialized to 2. This `a` does not conflict with the `a` in `main`, since the visibility of **static** variables at the internal level is restricted to the block in which they are declared.

The variable `a` is increased by 2, giving 4 as the result. If the `other` function were called again in the same program, the initial value of `a` would be 4. Internal **static** variables retain their values when the block in which they are declared is exited and reentered.

4.6.3 Function Declarations at the External and Internal Levels

Function declarations can use either the **static** or the **extern** storage-class specifier. Functions always have global lifetimes.

The visibility rules for functions are slightly different from the rules for variables. Function declarations at the internal level have the same meaning as function declarations at the external level. This means that functions cannot have block visibility, and the visibility of functions cannot be nested. A function declared to be **static** is visible only within the source file in which it is defined. Any function in the same source file can call the **static** function, but functions in other source files cannot. Another **static** function by the same name can be declared in a different source file without conflict.

Functions declared as **extern** are visible throughout all the source files that constitute the program (unless they are later redeclared as **static**). Any function can call an **extern** function.

Function declarations that omit the storage-class specifier default to **extern**.

4.7 Initialization

A variable can be set to an initial value by applying an initializer to the declarator in the variable declaration. The value or values of the initializer are assigned to the variable. The initializer is preceded by an equal sign (=), as shown below:

= *initializer*

Variables of any type can be initialized, with the restrictions outlined below. Functions do not take initializers.

Declarations that use the **extern** storage-class specifier cannot contain initializers.

Variables declared at the external level can be initialized; if not explicitly initialized, they are set to 0 at compile time. Any variable declared with the **static** storage-class specifier can be initialized with a constant expression. Initializations of **static** variables are performed once, at compile time. If not explicitly initialized, **static** variables are automatically set to 0.

Initializations of **auto** and **register** variables are performed each time execution control passes to the block in which they are declared. If the initializer is omitted from the declaration of an **auto** or **register** variable, the initial value of the variable is undefined.

Initializations of **auto** aggregate types (arrays, structures, and unions) are prohibited. Only **static** aggregates and aggregates declared at the external level can be initialized.

The initial values for external variable declarations and for all **static** variables, whether external or internal, must be constant expressions. Constant expressions are described in Section 5.2.10. Automatic and register variables can be initialized with either constant or variable values.

Sections 4.7.1 and 4.7.2 describe how to initialize variables of fundamental, pointer, and aggregate types.

4.7.1 Fundamental and Pointer Types

Syntax

= *expression*

The value of *expression* is assigned to the variable. The conversion rules for assignment apply.

Examples

```
int x = 10;                    /* Example 1 */

register int *px = 0;          /* Example 2 */

int c = (3 * 1024);            /* Example 3 */

int *b = &x;                   /* Example 4 */
```

In the first example, `x` is initialized to the constant expression 10. In the second example, the pointer `px` is initialized to 0, producing a "null" pointer. The third example uses a constant expression to initialize `c`. The fourth example initializes the pointer `b` with the address of another variable, `x`.

4.7.2 Aggregate Types

Syntax

= { *initializer-list*}

An *initializer-list* is a list of initializers separated by commas. Each initializer in the list is either a constant expression or an *initializer-list*. Therefore, a brace-enclosed list can appear within another *initializer-list*. This is useful for initializing aggregate members of an aggregate, as shown in the examples below.

For each *initializer-list*, the values of the constant expressions are assigned in order to the members of the aggregate variable. When a union is initialized, the *initializer-list* must be a single constant expression. The value of the constant expression is assigned to the first member of the union.

If there are fewer values in an *initializer-list* than there are in the aggregate type, the remaining members or elements are initialized to 0. Giving too many initial values for the aggregate type causes an error. These rules apply to each embedded *initializer-list*, as well as to the aggregate as a whole.

For example,

```
int P[4][3] = {
        { 1, 1, 1 },
        { 2, 2, 2 },
        { 3, 3, 3,},
        { 4, 4, 4,},
};
```

declares `P` as a 4-by-3 array and initializes the elements of its first row to 1, the elements of its second row to 2, and so on through the fourth row. Note that the *initializer-list* for the third and fourth rows contains commas after the last constant expression. The last *initializer-list* (`{4, 4, 4,}`) is also followed by a comma. These extra commas are permitted but are not required; the required commas are those that separate constant expressions and *initializer-lists*.

If there is no embedded initializer list for an aggregate member, values are simply assigned, in order, to each member of the subaggregate. Therefore, the above initialization is equivalent to the following:

```
int P[4][3] = {
        1, 1, 1, 2, 2, 2, 3, 3, 3, 4, 4, 4
};
```

Braces can also appear around individual initializers in the list.

When initializing aggregate variables, care must be taken to use braces and initializer lists properly. The following example illustrates in more detail the compiler's interpretation of braces:

```
typedef struct {
    int n1, n2, n3;
} triplet;

triplet nlist[2][3] = {
    { {  1, 2, 3 }, {  4, 5, 6 }, {  7, 8, 9 } },   /* Line 1 */
    { { 10,11,12 }, { 13,14,15 }, { 15,16,17 } }    /* Line 2 */
};
```

In the above example, `nlist` is declared as a 2-by-3 array of structures, each structure having three members. Line 1 of the initialization assigns values to the first row of `nlist`, as follows:

1. The first left brace on Line 1 signals the compiler that initialization of the first aggregate member of `nlist` is beginning (that is, `nlist[0]`).
2. The second left brace indicates that initialization of the first aggregate member of `nlist[0]` is beginning (that is, the structure at `nlist[0][0]`).
3. The first right brace ends initialization of the structure `nlist[0][0]`; the next left brace starts initialization of `nlist[0][1]`.
4. The process continues to the end of the line, where the closing right brace ends initialization of `nlist[0]`.

Similarly, Line 2 assigns values to the second row of `nlist`.

Note that the outer sets of braces enclosing the initializers on Line 1 and on Line 2 are required. The following construction, which omits the outer braces, would cause an error:

```
        /* THIS CAUSES AN ERROR */
triplet nlist[2][3] = {
        {  1, 2, 3 },{  4, 5, 6 },{  7, 8, 9 },    /* Line 1 */
        { 10,11,12 },{ 13,14,15 },{ 16,17,18 }     /* Line 2 */
};
```

In the above construction, the first left brace on Line 1 starts the initialization of `nlist[0]`, which is an array of three structures. The values 1, 2, and 3 are assigned to the three members of the first structure. When the next right brace is encountered (after the value 3), initialization of `nlist[0]` is complete, and the two remaining structures in the three-structure array are automatically initialized to 0. Similarly, `{ 4,5,6 }` initializes the first structure in the second row of `nlist`, and the remaining two structures of `nlist[1]` are set to 0. When the compiler encounters the next initializer list (`{ 7,8,9 }`), it attempts to initialize `nlist[2]`. Since `nlist` has only two rows, this produces an error.

Examples

```
/********************* Example 1 *********************/

struct list {
        int i, j, k;
        float m[2][3];
        } x = {
                1,
                2,
                3,
                {4.0, 4.0, 4.0}
        };

/********************* Example 2 *********************/

union {
        char x[2][3];
        int i, j, k;
        } y = {
                {'1'},
                {'4'}
        };
```

In the first example, the three **int** members of `x` are initialized to 1, 2, and 3, respectively. The three elements in the first row of `m` are initialized to 4.0; the elements of the remaining row of `m` are initialized to 0 by default.

In the second example, the union variable `y` is initialized. The first element of the union is an array, so the initializer is an aggregate initializer. The initializer list `{'1'}` gives values to the first row of the array. Since only one value appears in the list, the element in the first column is initialized to the character `1`, and the remaining two elements in the row are initialized to 0 (the null character), by default. Similarly, the first element of the second row of `x` is initialized to the character `4`, and the remaining two elements in the row are initialized to 0.

4.7.3 String Initializers

An array can be initialized with a string literal. For example,

```
char code[ ] = "abc";
```

initializes `code` as a four-element array of characters. The fourth element is the null character that terminates all string literals.

If the array size is specified and the string is longer than the specified size of the array, the extra characters are simply discarded. The following declaration initializes `code` as a three-element character array:

```
char code[3] = "abcd";
```

Only the first three characters of the initializer are assigned to `code`. The character `d` and the null character are discarded.

If the string is shorter than the specified size of the array, the remaining elements of the array are initialized to 0 (the null character).

4.8 Type Declarations

A type declaration defines the name and members of a structure or union type, or the name and enumeration set of an enumeration type. The name of a declared type can be used in variable or function declarations to refer to that type. This is useful if many variables and functions have the same type.

A **typedef** declaration defines a type specifier for a type. These declarations are used to construct shorter or more meaningful names for types already defined by C or for types declared by the user.

4.8.1 Structure, Union, and Enumeration Types

Declarations of structure, union, and enumeration types have the same general form as variable declarations of those types. In type declarations the variable identifier is omitted, since no variable is declared. The *tag* is mandatory; it names the structure, union, or enumeration type. The *member-declaration-list* or *enum-list* defining the type must appear in the type declaration; the abbreviated form of variable declarations, in which a *tag* refers to a type defined elsewhere, is not legal for type declarations.

Examples

```
/********************* Example 1 *********************/

enum status {
        loss = -1,
        bye,
        tie = 0,
        win
        };

/********************* Example 2 *********************/

struct student {
        char name[20];
        int id, class;
        };
```

The first example declares an enumeration type named `status`. The name of the type can be used in declarations of enumeration variables. The identifier `loss` is explicitly set to −1. Both `bye` and `tie` are associated with the value 0, and `win` is given the value 1.

The second example declares a structure type named `student`. A declaration such as `struct student employee;` can be used to declare a structure variable with `student` type.

4.8.2 Typedef Declarations

Syntax

typedef *type-specifier declarator* [[, *declarator*...]];

A **typedef** declaration is analogous to a variable declaration except that the **typedef** keyword replaces a storage-class specifier. The declaration is interpreted in the same way as variable and function declarations, but the identifier, instead of assuming the type specified by the declaration, becomes a synonym for the type.

A **typedef** declaration does not create types. It creates synonyms for existing types, or names for types that could be specified in other ways. Any type can be declared with **typedef**, including pointer, function, and array types. A **typedef** name for a pointer to a structure or union type can be declared before the structure or union type is defined, as long as the definition has the same visibility as the declaration.

Examples

```
/********************* Example 1 *********************/

typedef int WHOLE;

/********************* Example 2 *********************/

typedef struct club {
        char name[30];
        int size, year;
        } GROUP ;

/********************* Example 3 *********************/

typedef GROUP *PG;

/********************* Example 4 *********************/

typedef void DRAWF(int, int);
```

The first example declares WHOLE **to be a synonym for int.**

The second example declares GROUP as a structure type with three members. Since a structure tag, club, is also specified, either the **typedef** name (GROUP) or the structure tag can be used in declarations.

The third example uses the previous **typedef** name to declare a pointer type. The type PG is declared as a pointer to the GROUP type, which in turn is defined as a structure type.

The final example provides the type DRAWF for a function returning no value and taking two **int** arguments. This means, for example, that the declaration DRAWF box; is equivalent to the declaration void box(int, int);.

4.9 Type Names

A "type name" specifies a particular data type. Type names are used in three contexts: in the argument-type lists of function declarations, in type casts, and in **sizeof** operations. Argument-type lists are discussed in Section 4.5, "Function Declarations." Type casts and **sizeof** operations are discussed in sections 5.7.2 and 5.3.4, respectively.

The type names for fundamental, enumeration, structure, and union types are simply the type specifiers for those types.

A type name for a pointer, array, or function type has the following form:

type-specifier abstract-declarator

An *abstract-declarator* is a declarator without an identifier, consisting of one or more pointer, array, or function modifiers. The pointer modifier (*) always appears before the identifier in a declarator, while array ([ ]) and function (( )) modifiers appear after the identifier. It is thus possible to determine where the identifier would appear in an abstract declarator and interpret the declarator accordingly.

Abstract declarators can be complex. Parentheses in a complex abstract declarator specify a particular interpretation, just as they do for the complex declarators in declarations.

The abstract declarator consisting of a set of empty parentheses, ( ), is not allowed because it is ambiguous. It is impossible to determine whether the

implied identifier belongs inside the parentheses, in which case it is an unmodified type, or before the parentheses, in which case it is a function type.

The type specifiers established through **typedef** declarations also qualify as type names.

Examples

```
long *                          /* Example 1 */

int (*)[5]                      /* Example 2 */

int (*)(void)                   /* Example 3 */
```

The first example gives the type name for "pointer to **long**" type.

The second and third examples show how parentheses modify complex abstract declarators. Example 2 gives the type name for a pointer to an array of five **int** values. Example 3 names a pointer to a function taking no arguments and returning an **int**.

Chapter 5
Expressions and Assignments

5.1 Introduction 87
5.2 Operands 87
5.2.1 Constants 88
5.2.2 Identifiers 88
5.2.3 Strings 89
5.2.4 Function Calls 89
5.2.5 Subscript Expressions 90
5.2.6 Member-Selection Expressions 92
5.2.7 Expressions with Operators 94
5.2.8 Expressions in Parentheses 95
5.2.9 Type-Cast Expressions 95
5.2.10 Constant Expressions 95
5.3 Operators 96
5.3.1 Usual Arithmetic Conversions 97
5.3.2 Complement Operators 98
5.3.3 Indirection and Address-of Operators 99
5.3.4 The sizeof Operator 101
5.3.5 Multiplicative Operators 102
5.3.6 Additive Operators 104
5.3.7 Shift Operators 107
5.3.8 Relational Operators 108
5.3.9 Bitwise Operators 110
5.3.10 Logical Operators 112
5.3.11 Sequential-Evaluation Operator 114

5.3.12 Conditional Operator 115
5.4 Assignment Operators 116
5.4.1 Lvalue Expressions 116
5.4.2 Unary Increment and Decrement 117
5.4.3 Simple Assignment 118
5.4.4 Compound Assignment 118
5.5 Precedence and Order of Evaluation 120
5.6 Side Effects 123
5.7 Type Conversions 124
5.7.1 Assignment Conversions 124
5.7.1.1 Conversions from Signed Integral Types 124
5.7.1.2 Conversions from Unsigned Integral Types 126
5.7.1.3 Conversions from Floating-Point Types 128
5.7.1.4 Conversions to and from Pointer Types 129
5.7.1.5 Conversions from Other Types 130
5.7.2 Type-Cast Conversions 130
5.7.3 Operator Conversions 130
5.7.4 Function-Call Conversions 131

5.1 Introduction

This chapter describes how to form expressions and make assignments in the C language. An expression is a combination of operands and operators that yields ("expresses") a single value. An operand is a constant or variable value that is manipulated in the expression. Each operand of an expression is also an expression, since it represents a single value. Operators specify how the operand or operands of the expression are manipulated.

In C, assignments are considered expressions. An assignment yields a value; its value is the value being assigned. In addition to the simple assignment operator (=), C offers complex assignment operators that both transform and assign their operands.

The value resulting from an expression's evaluation depends on the relative precedence of operators in the expression and on side effects, if present. The precedence of operators determines the grouping of operands in an expression. Side effects are changes caused by the evaluation of an expression. In an expression with side effects, the evaluation of one operand can affect the value of another. With some operators, the order in which operands are evaluated also affects the result of the expression.

The value represented by each operand in an expression has a type, which may be converted to a different type in certain contexts. Type conversions occur in assignments, type casts, function calls, and operations.

5.2 Operands

A C operand is a constant, an identifier, a string, a function call, a subscript expression, a member-selection expression, or a more complex expression formed by combining operands with operators or enclosing operands in parentheses. Any operand that yields a constant value is called a "constant expression."

Every operand has a type. The following sections discuss the type of value each kind of operand represents. An operand can be cast from its original type to another type by means of a "type-cast" operation. A type-cast expression can also form an operand of an expression.

5.2.1 Constants

A constant operand has the value and type of the constant value it represents. A character constant has **int** type. An integer constant has **int**, **long**, **unsigned int**, or **unsigned long** type, depending on the integer's size and how the value is specified. Floating-point constants always have **double** type. String literals are considered arrays of characters and are discussed in Section 5.2.3.

5.2.2 Identifiers

An identifier names a variable or function. Every identifier has a type, which is established when the identifier is declared. The value of an identifier depends on its type, as follows:

- Identifiers of integral and floating-point types represent values of the corresponding type.

- An identifier of **enum** type represents one constant value of a set of constant values. The value of the identifier is the constant value. Its type is **int**, by definition of the **enum** type.

- An identifier of **struct** or **union** type represents a value of the specified **struct** or **union** type.

- An identifier declared as a pointer represents a pointer to the specified type.

- An identifier declared as an array represents a pointer whose value is the address of the first element of the array. The type addressed by the pointer is the type of the elements of the array. For example, if `series` is declared to be a 10-element integer array, the identifier `series` expresses the address of the array, while the subscript expression `series[5]` refers to a variable integer element of `series`. Subscript expressions are discussed in Section 5.2.5.

  The address of an array does not change during the execution of the program, although the values of the individual elements can change. The pointer value represented by an array identifier is not a variable, and an array identifier cannot form the left-hand operand of an assignment operation.

- An identifier declared as a function represents a pointer whose value is the address of the function. The type addressed by the pointer is a function returning a value of a specified type. The address of a

function does not change during the execution of a program; only the return value varies. Thus, function identifiers cannot be left-hand operands in assignment operations.

5.2.3 Strings

A string literal consists of a list of characters enclosed in double quotes, as shown below:

"*string*"

A string literal is stored as an array of elements with **char** type. The string literal represents the address of the first element of the array. The address of the string's first element is a constant, so the value represented by a string expression is a constant.

Since string literals are effectively pointers, they can be used in contexts that allow pointer values, and they are subject to the same restrictions as pointers. String literals have one additional restriction: they are not variables and cannot be left-hand operands in assignment operations.

The last character of a string is always the null character, **\0**. The null character is not visible in the string expression, but it is added as the last element when the string is stored. Therefore, the string `"abc"` actually has four characters rather than three.

5.2.4 Function Calls

Syntax

expression (*expression-list*)

A function call consists of an *expression* followed by an *expression-list* in parentheses, where *expression* evaluates to a function address (for example, a function identifier), and *expression-list* is a list of expressions (separated by commas) whose values, the actual arguments, are passed to the function. The *expression-list* can be empty.

A function-call expression has the value and type of the function's return value. If the function's return type is **void**, the function-call expression also has **void** type. If control returns from the called function without execution of a **return** statement, the value of the function call is undefined. See Section 7.4 for more information about function calls.

5.2.5 Subscript Expressions

Syntax

expression1 [*expression2*]

A subscript expression represents the value at the address that is *expression2* positions beyond *expression1*. *Expression1* is any pointer value, such as an array identifier, and *expression2* is an integral value. *Expression2* must be enclosed in brackets ([ ]).

Subscript expressions are generally used to refer to array elements, but a subscript can be applied to any pointer.

The subscript expression is evaluated by adding the integral value (*expression2*) to the pointer value (*expression1*), then applying the indirection operator (*) to the result. (See Section 5.3.3 for a discussion of the indirection operator.) In effect, for a one-dimensional array, the following four expressions are equivalent, assuming that `a` is a pointer and `b` is an integer.

```
a[b]
*(a + b)
*(b + a)
b[a]
```

According to the conversion rules of the addition operator (see Section 5.3.6), the integral value is converted to an address offset by multiplying it by the length of the type addressed by the pointer.

For example, suppose the identifier `line` refers to an array of **int** values. To evaluate the expression `line[i]`, the integer value `i` is multiplied by the length of an **int**. The converted value of `i` represents `i` **int** positions. This converted value is added to the original pointer value (`line`) to yield an address that is offset `i` **int** positions from `line`.

As the last step in evaluating the subscript expression, the indirection operator is applied to the new address. The result is the value of the array element at that position (intuitively, `line[i]`).

Note that the subscript expression

```
line[0]
```

represents the value of the first element of `line`, since the offset from the

address represented by `line` is 0. Similarly, an expression such as

```
line[5]
```

refers to the element offset five positions from `line`, or the sixth element of the array.

Multidimensional-Array References

A subscript expression can be subscripted, as follows:

expression1 [*expression2*] [*expression3*]...

Subscript expressions associate left to right. The leftmost subscript expression, *expression1*[*expression2*], is evaluated first. The address that results from adding *expression1* and *expression2* forms the pointer expression to which *expression3* is added. The indirection operator (*) is applied after the last subscripted expression is evaluated. However, the indirection operator is not applied if the final pointer value addresses an array type (see the third example below).

Expressions with multiple subscripts refer to elements of multidimensional arrays. A multidimensional array is an array whose elements are arrays. The first element of a three-dimensional array, for example, is an array with two dimensions.

Examples

```
int prop[3][4][6];
int i, *ip, (*ipp)[6];

i = prop[0][0][1];              /* Example 1 */

i = prop[2][1][3];              /* Example 2 */

ip = prop[2][1];                /* Example 3 */

ipp = prop[2];                  /* Example 4 */
```

The array named `prop` has 3 elements, each of which is a 4-by-6 array of **int** values.

Example 1 shows how to refer to the second individual **int** element of `prop`. Arrays are stored by row, so the last subscript varies the most quickly; the expression `prop[0][0][2]` refers to the next (third) element in the array, and so on.

The second example shows a more complex reference to an individual element of `prop`. The expression is evaluated as follows:

1. The first subscript, `2`, is multiplied by the size of a 4-by-6 **int** array and added to the pointer value `prop`. The result points to the third 4-by-6 array of `prop`.
2. Next, the second subscript, `1`, is multiplied by the size of the 6-element **int** array and added to the address represented by `prop[2]`.
3. Each element of the 6-element array is an **int** value, so the final subscript, `3`, is multiplied by the size of an **int** before it is added to `prop[2][1]`. The resulting pointer addresses the fourth element of the 6-element array.
4. The last step in evaluating the expression `prop[2][1][3]` is applying the indirection operator to the pointer value. The result is the **int** element at that address.

Examples 3 and 4 show cases where the indirection operator is not applied. In Example 3, the expression `prop[2][1]` is a valid reference to the three-dimensional array `prop`; it refers to a 6-element array. Since the pointer value addresses an array, the indirection operator is not applied. Similarly, the result of the expression `prop[2]` in Example 4 is a pointer value addressing an array with two dimensions.

5.2.6 Member-Selection Expressions

Syntax

expression.identifier
expression–>identifier

Member-selection expressions refer to members of structures and unions. A member-selection expression has the value and type of the selected member.

In the first form, *expression.identifier*, *expression* represents a value of **struct** or **union** type, and the *identifier* names a member of the specified structure or union.

In the second form, *expression*−>*identifier*, *expression* represents a pointer to a structure or union, and the *identifier* names a member of the specified structure or union.

The two forms of member-selection expressions have similar effects. In fact, expressions involving the pointer selection operator (−>) are shorthand versions of expressions using the period (.) in cases in which the expression before the period consists of the indirection operator (*) applied to a pointer value. (The indirection operator is discussed in Section 5.3.3.) Therefore,

expression−>*identifier*

is equivalent to

(**expression*).*identifier*

when *expression* is a pointer value.

Examples

```
struct pair {
        int a;
        int b;
        struct pair *sp;
        } item, list[10];

item.sp = &item;                    /* Example 1 */

(item.sp)->a = 24;                  /* Example 2 */

list[8].b = 12;                     /* Example 3 */
```

In the first example, the address of the `item` structure is assigned to the `sp` member of the structure. This means that `item` contains a pointer to itself.

In the second example, the pointer expression `item.sp` is used with the pointer selection operator (−>) to assign a value to the member `a`.

The third example shows how to select an individual structure member from an array of structures.

5.2.7 Expressions with Operators

Expressions with operators can be unary, binary, or ternary expressions. A unary expression consists of an operand prefixed by a unary operator ("unop") or an operand enclosed in parentheses and preceded by the **sizeof** keyword:

unop operand
sizeof (*operand*)

A binary expression consists of two operands joined by a binary operator ("binop"):

operand binop operand

A ternary expression consists of three operands joined by the ternary (**? :**) operator:

operand **?** *operand* **:** *operand*

Assignment expressions use unary or binary assignment operators. The unary assignment operators are the increment (++) and decrement (--) operators; the binary assignment operators are the simple assignment operator (=) and the compound assignment operators (referred to as "compound-assign-ops"). Each compound assignment operator is a combination of another binary operator with the simple assignment operator. The forms of assignment expressions are as follows:

operand++
operand--
++*operand*
--*operand*
operand = *operand*
operand compound-assignment-op operand

5.2.8 Expressions in Parentheses

Any operand can be enclosed in parentheses; they have no effect on the type or value of the enclosed expression. For example, in the expression

```
(10 + 5) / 5
```

the parentheses around `10 + 5` mean that the value of `10 + 5` is the left operand of the division (/) operator. The result of `(10 + 5) / 5` is 3. Without the parentheses, `10 + 5 / 5` would evaluate to 11.

Although parentheses affect the way operands are grouped in an expression, they cannot guarantee a particular order of evaluation for the expression.

5.2.9 Type-Cast Expressions

A type-cast expression has the following form:

(*type-name*) *operand*

Type-cast conversions are discussed in Section 5.7.2; type names are discussed in Section 4.9.

5.2.10 Constant Expressions

A constant expression is any expression that evaluates to a constant. The operands of a constant expression can be integer constants, character constants, floating-point constants, enumeration constants, type casts to integral and floating-point types, and other constant expressions. The operands can be combined and modified using operators, as described in Section 5.2.7, with some restrictions.

Constant expressions cannot use assignment operators (see Section 5.4) or the binary sequential evaluation operator (,). The unary address-of operator (&) can be used only in certain initializations (see the last paragraph of this section, 5.2.10).

Constant expressions used in preprocessor directives are subject to additional restrictions, and are consequently known as *restricted-constant-expressions.* A *restricted-constant-expression* cannot contain **sizeof** expressions, enumeration constants, or type casts to any type. It can, however, contain the special constant expression **defined**(*identifier*). See Section 8.2.1, "The #define Directive," for more information.

These additional restrictions also apply to constant expressions used to initialize variables at the external level. However, such expressions are allowed to apply the unary address-of operator (&) to other external-level variables with fundamental, structure, and union types and to external-level arrays subscripted with a constant expression. In these expressions, a constant expression not involving the address-of operator can be added to or subtracted from the address expression.

5.3 Operators

C operators take one operand (unary operators), two operands (binary operators), or three operands (the ternary operator). Assignment operators are unary or binary operators; the assignment operators are described in Section 5.4.

Unary operators prefix their operand and associate right to left. C's unary operators are as follows:

Symbol	Name
– ~ !	Complement operators
* &	Indirection and address-of operators
sizeof	Size operator

Binary operators associate left to right. The binary operators are as follows:

Symbol	Name
* / %	Multiplicative operators
+ –	Additive operators
<< >>	Shift operators
< > <= >= == !=	Relational operators
& | ^	Bitwise operators
&& ||	Logical operators
,	Sequential-evaluation operator

C has one ternary operator, the conditional operator (**? :**). It associates right to left.

5.3.1 Usual Arithmetic Conversions

Most C operators perform type conversions to bring the operands of an expression to a common type or to extend short values to the integer size used in machine operations. The conversions performed by C operators depend on the specific operator and the type of the operand or operands. However, many operators perform similar conversions on operands of integral and floating-point types. These conversions are known as "arithmetic" conversions because they apply to the types of values ordinarily used in arithmetic.

The arithmetic conversions summarized below are called the "usual arithmetic conversions." The discussion of each operator in the following sections specifies whether or not the operator performs the usual arithmetic conversions. It also specifies the additional conversions, if any, the operator performs.

The specific path of each type of conversion is outlined in Section 5.7.

The usual arithmetic conversions proceed in the following order:

1. Any operands of **float** type are converted to **double** type.
2. If one operand has **double** type, the other operand is converted to **double**.
3. Any operands of **char** or **short** type are converted to **int**.
4. Any operands of **unsigned char** or **unsigned short** type are converted to **unsigned int** type.
5. If one operand is of type **unsigned long**, the other operand is converted to **unsigned long**.
6. If one operand is of type **long**, the other operand is converted to **long**.
7. If one operand is of type **unsigned int**, the other operand is converted to **unsigned int**.

5.3.2 Complement Operators

Arithmetic Negation (–)

The arithmetic-negation operator (–) produces the negative (two's complement) of its operand. The operand must be an integral or floating-point value. The usual arithmetic conversions are performed.

Bitwise Complement (~)

The bitwise-complement operator (~) produces the bitwise complement of its operand. The operand must be of integral type. The usual arithmetic conversions are performed; the result has the type of the operand after conversion.

Logical-NOT (!)

The logical-NOT operator (!) produces the value 0 if its operand is true (nonzero) and the value 1 if its operand is false (0). The result has **int** type. The operand must be an integral, floating-point, or pointer value.

Examples

```
/********************* Example 1 *********************/
short x = 987;
    x = -x;

/********************* Example 2 *********************/
unsigned short y = 0xaaaa;
    y = ~y;

/********************* Example 3 *********************/
if ( !(x < y));
```

In the first example, the new value of x is the negative of 987, or −987.

In the second example, the new value assigned to y is the one's complement of the unsigned value 0xaaaa, or 0x5555.

In the third example, if x is greater than or equal to y, the result of the expression is 1 (true). If x is less than y, the result is 0 (false).

5.3.3 Indirection and Address-of Operators

Indirection (*)

The indirection operator (*) accesses a value indirectly, through a pointer. The operand must be a pointer value. The result of the operation is the value to which the operand points. The result type is the type addressed by the pointer operand. If the pointer value is null, the result is unpredictable.

Address-of (&)

The address-of operator (&) takes the address of its operand. The operand can be any value that can appear as the left-hand value of an assignment operation. (Assignment operations are discussed in Section 5.4.) The result of the address operation is a pointer to the operand. The type addressed by the pointer is the type of the operand.

The address-of operator cannot be applied to a bit-field member of a structure, nor can it be applied to an identifier declared with the **register** storage-class specifier.

Examples

```
int *pa, x;
int a[20];
double d;

pa = &a[5];                 /* Example 1 */

x = *pa;                    /* Example 2 */

if (x == *&x)               /* Example 3 */
    printf("True\n");

d = *(double *)(&x);        /* Example 4 */
```

In the first example, the address-of operator (&) takes the address of the sixth element of the array `a`. The result is stored in the pointer variable `pa`.

The indirection operator (*) is used in the second example to access the **int** value at the address stored in `pa`. The value is assigned to the integer variable `x`.

In Example 3, the word `True` would be printed. This example demonstrates that the result of applying the indirection operator to the address of `x` is the same as `x`.

Example 4 shows a useful application of the rule shown in Example 3. The address of `x` is converted by a type cast to a pointer to a **double**; the indirection operator is then applied, and the result of the expression is a **double** value.

5.3.4 The sizeof Operator

The **sizeof** operator determines the amount of storage associated with an identifier or a type. A **sizeof** expression has the form

sizeof(*name*)

where *name* is either an identifier or a type name. The type name may not be **void**. The value of a **sizeof** expression is the amount of storage, in bytes, associated with the named identifier or type.

When the **sizeof** operator is applied to an array identifier, the result is the size of the entire array in bytes rather than the size of the pointer represented by the array identifier.

When the **sizeof** operator is applied to a structure or union type name, or to an identifier of structure or union type, the result is the actual size in bytes of the structure or union, which may include internal and trailing padding used to align the members of the structure or union on memory boundaries. Thus, the result may not correspond to the size calculated by adding up the storage requirements of the members.

Example

```
buffer = calloc(100, sizeof (int) );
```

With the **sizeof** operator you can avoid specifying machine-dependent data sizes in your program. The above example uses the **sizeof** operator to pass the size of an **int**, which varies across machines, as an argument to a function named `calloc`. The value returned by the function is stored in `buffer`.

5.3.5 Multiplicative Operators

The multiplicative operators perform multiplication (*), division (/), and remainder (%) operations. The operands of the remainder operator (%) must be integral; the multiplication (*) and division (/) operators take integral and floating-point operands. The types of the operands can be different. The multiplicative operators perform the usual arithmetic conversions on the operands. The type of the result is the type of the operands after conversion.

The conversions performed by the multiplicative operators make no provision for overflow or underflow conditions. Information is lost if the result of a multiplicative operation cannot be represented in the type of the operands after conversion.

Multiplication (*)

The multiplication operator (*) specifies that its two operands are to be multiplied.

Division (/)

The division operator (/) specifies that its first operand is to be divided by the second. When two integers are divided, the result, if not an integer, is truncated. If both operands are positive or unsigned, the result is truncated toward 0. The direction of truncation when either operand is negative may be either toward or away from 0, depending on the implementation. Division by 0 gives unpredictable results.

Remainder (%)

The result of the remainder operator (%) is the remainder when the first operand is divided by the second.

Examples

```
int i = 10, j = 3, n;
double x = 2.0, y;

y = x * i;                    /* Example 1 */

n = i / j;                    /* Example 2 */

n = i % j;                    /* Example 3 */
```

In the first example, `x` is multiplied by `i` to give the value 20.0. The result has **double** type.

In the second example, 10 is divided by 3. The result is truncated toward 0, yielding the integer value 3.

In the third example, `n` is assigned the integer remainder 1 when 10 is divided by 3.

5.3.6 Additive Operators

The additive operators perform addition (+) and subtraction (–). The operands can be integral or floating-point values; some additive operations can also be performed on pointer values, as outlined under the discussion of each operator. The usual arithmetic conversions are performed on integral and floating-point operands. The type of the result is the type of the operands after conversion.

The conversions performed by the additive operators make no provision for overflow or underflow conditions. Information is lost if the result of an additive operation cannot be represented in the type of the operands after conversion.

Addition (+)

The addition operator (+) specifies addition of its two operands. The operands can have integral or floating-point types, as described previously, or one operand can be a pointer and the other an integer. When an integer is added to a pointer, the integer value (*i*) is converted by multiplying it by the length of the value addressed by the pointer. After conversion, the integer value represents *i* memory positions, where each position has the length specified by the pointer type. When the converted integer value is added to the pointer value, the result is a new pointer value expressing the address *i* positions from the original address. The new pointer value addresses the same type as the original pointer value.

Subtraction (–)

The subtraction operator (–) subtracts its second operand from the first. The operands can be integral or floating-point values, as described earlier. The subtraction operator also allows the subtraction of an integer from a pointer value and the subtraction of two pointer values.

When an integer value is subtracted from a pointer value, the same conversions occur as with addition of a pointer and integer. The subtraction operator converts the integer value with respect to the type addressed by the pointer value. The result is the memory address *i* positions before the original address, where *i* is the integer value and each position is the length of the type addressed by the pointer value. The new pointer points to the type addressed by the original pointer value.

Two pointer values can be subtracted if they point to the same type. The difference between the two pointers is converted to a signed integer value by dividing the difference by the length of the type the pointers address. The result represents the number of memory positions of that type between the two addresses. The result is only guaranteed to be meaningful for two elements of the same array, as discussed next.

Pointer Arithmetic

Additive operations involving a pointer and an integer generally give meaningful results only when the pointer operand addresses an array member and the integer value produces an offset within the bounds of the same array. The conversion of the integer value to an address offset assumes that only memory positions of the same size lie between the original address and the address plus offset.

The preceding assumption is valid for array members. An array is by definition a series of values of the same type; its elements reside in contiguous memory locations. Storage for any types except array elements is not guaranteed to be completely filled. That is, blanks can occur between memory positions, even positions of the same type. Adding to or subtracting from addresses referring to any values but array elements gives unpredictable results.

Similarly, the conversion involved in the subtraction of two pointer values assumes that only values of the same type, with no blanks, lie between the two addresses given by the operands.

Additive operations between pointer and integer values on machines with segmented architecture (such as the 8086/8088) may not be valid in some cases. See your system documentation for more information.

Examples

```
int i = 4, j;
float x[10];
float *px;

px = &x[4] + i;          /* Example 1 */

j = &x[i] - &x[i-2];     /* Example 2 */
```

In the first example, the integer operand `i` is added to the address of the fifth element of `x`. The value of `i` is multiplied by the length of a **float** and added to `&x[4]`. The resulting pointer value is the address of `x[8]`.

In the second example, the address of the third element of `x` (given by `x[i-2]`) is subtracted from the address of the fifth element of `x` (given by `x[i]`). The difference is divided by the length of a **float**; the result is the integer value 2.

5.3.7 Shift Operators

The shift operators shift their first operand left (<<) or right (>>) by the number of positions the second operand specifies. Both operands must be integral values. The usual arithmetic conversions are performed; the type of the result is the type of the left operand after conversion.

For leftward shifts, the vacated right bits are set to 0. In a rightward shift, the method of filling left bits depends on the type, after conversion, of the first operand. If the type is **unsigned**, vacated left bits will be set to 0. Otherwise, vacated left bits are filled with copies of the sign bit.

The result of a shift operation is undefined if the second operand is negative.

The conversions performed by the shift operators make no provision for overflow or underflow conditions. Information is lost if the result of a shift operation cannot be represented in the type of the first operand after conversion.

Example

```
unsigned int x, y, z;

x = 0x00aa;
y = 0x5500;

z = (x << 8) + (y >> 8);
```

In the above example, `x` is shifted left by eight positions and `y` is shifted right eight positions. The shifted values are added, giving `0xaa55`, and assigned to `z`.

5.3.8 Relational Operators

The binary relational operators test their first operand against the second to determine if the relationship specified by the operator holds true. The result of a relational expression is 1 if the tested relationship holds and 0 if it does not. The type of the result is **int**. The relational operators test the following relationships:

Operator	Relationship Tested
<	First operand less than second operand
>	First operand greater than second operand
<=	First operand less than or equal to second operand
>=	First operand greater than or equal to second operand
==	First operand equal to second operand
!=	First operand not equal to second operand

The operands can have integral, floating-point, or pointer type. The types of the operands can be different. The usual arithmetic conversions are performed on integral and floating-point operands.

One or both operands of the equality (==) and inequality (!=) operators can have **enum** type; an **enum** value is converted in the same manner as an **int** value.

The operands of any relational operator can be two pointers to the same type. For the equality (==) and inequality (!=) operators, the result of the comparison reflects whether or not the two pointers address the same memory location. The result of pointer comparisons involving the other operators (<, >, <=, >=) reflects the relative position of two memory addresses.

Since the address of a given value is arbitrary, comparisons between the addresses of two unrelated values are generally meaningless. Comparisons between the addresses of different elements of the same array can be useful, however, since array elements are guaranteed to be stored in order from the first element to the last. The address of the first array element is "less than" the address of the last element.

A pointer value can be compared for equality (==) or inequality (!=) to the constant value 0. A pointer with a value of 0 does not point to a memory location: it is called a "null" pointer. A pointer value is equal to 0 only if it is explicitly given that value through assignment or initialization.

Examples

```
int x = 0, y = 0;

x < y              /* Example 1 */

x > y              /* Example 2 */

x <= y             /* Example 3 */

x >= y             /* Example 4 */

x == y             /* Example 5 */

x != y             /* Example 6 */
```

When x and y are equal, expressions 3, 4, and 5 have the value 1 and expressions 1, 2, and 6 have the value 0.

5.3.9 Bitwise Operators

The bitwise operators perform bitwise-AND (&), inclusive-OR (|), and exclusive-OR (^) operations. The operands of bitwise operators must have integral type, but their types can be different. The usual arithmetic conversions are performed; the type of the result is the type of the operands after conversion.

Bitwise AND (&)

The bitwise-AND (&) operator compares each bit of its first operand to the corresponding bit of the second operand. If both bits are 1, the corresponding bit of the result is set to 1; otherwise, the corresponding result bit is set to 0.

Bitwise Inclusive OR (|)

The bitwise-inclusive-OR (|) operator compares each bit of its first operand to the corresponding bit of the second operand. If either of the compared bits is a 1, the corresponding bit of the result is set to 1. Otherwise, both bits are 0, and the corresponding result bit is set to 0.

Bitwise Exclusive OR (^)

The bitwise-exclusive-OR (^) operator compares each bit of its first operand to the corresponding bit of the second operand. If one of the compared bits is a 0 and the other bit is a 1, the corresponding bit of the result is set to 1; otherwise, the corresponding result bit is set to 0.

Examples

```
short i = 0xab00;
short j = 0xabcd;
short n;

n = i & j;                  /* Example 1 */

n = i | j;                  /* Example 2 */

n = i ^ j;                  /* Example 3 */
```

The result assigned to `n` in the first example is the same as `i` (AB00 hexadecimal). The bitwise inclusive OR in the second example results in the value ABCD (hexadecimal), while the bitwise exclusive OR in the third example produces CD (hexadecimal).

5.3.10 Logical Operators

The logical operators perform logical-AND (&&) and logical-OR (||) operations. The operands of the logical operators must have integral, floating-point, or pointer type. The types of the operands can be different.

The operands of logical-AND and logical-OR expressions are evaluated left to right. If the value of the first operand is sufficient to determine the result of the operation, the second operand is not evaluated.

Logical operators do not perform the standard arithmetic conversions. Instead, they evaluate each operand in terms of its equivalence to zero. A pointer has a value of 0 only if it is explicitly set to 0 through assignment or initialization.

The result of a logical operation is either 0 or 1, as described next. The type of the result is **int**.

Logical AND (&&)

The logical-AND operator (&&) produces the value 1 if both operands have nonzero values. If either operand is equal to 0, the result is 0. If the first operand of a logical-AND operation has a value of 0, the second operand is not evaluated.

Logical OR (||)

The logical-OR operator (||) performs an inclusive OR on its operands. It produces the value 0 if both operands have 0 values; if either operand has a nonzero value, the result is 1. If the first operand of a logical-OR operation has a nonzero value, the second operand is not evaluated.

Examples

```
int x, y;

if (x < y && y < z)                                    /* Example 1 */
        printf ("x is less than z\n");

if (x == y || x == z)                                  /* Example 2 */
        printf ("x is equal to either y or z\n");
```

In the first example, the `printf` function is called to print a message if `x` is less than `y` and `y` is less than `z`. If `x` is greater than `y`, the second operand (`y < z`) is not evaluated and nothing is printed. Note that this could cause problems in cases where the second operand contains side effects.

In the second example, a message is printed if `x` is equal to either `y` or `z`. If `x` is equal to `y`, the second operand (`x == z`) is not evaluated.

5.3.11 Sequential-Evaluation Operator

The sequential-evaluation operator (,) evaluates its two operands sequentially from left to right. The result of the operation has the value and type of the right operand. The types of the operands are unrestricted. No conversions are performed.

The sequential-evaluation operator (also called the "comma" operator) is typically used to evaluate two or more expressions in contexts that allow only one expression to appear.

Examples

```
/******************** Example 1 ********************/

for ( i = j = 1; i + j < 20; i += i, j--);

/******************** Example 2 ********************/

func_one(x, y + 2, z);
func_two((x--, y + 2), z);
```

In the first example, each operand of the **for** statement's third expression is evaluated independently. The left operand, `i += i,` is evaluated first, then `j--` is evaluated.

As shown in Example 2, the comma character is used in other contexts as a separator. In the function call to `func_one`, three arguments, separated by commas, are passed to the called function: `x`, `y + 2` and `z`. The use of the comma character as a separator must not be confused with its use as an operator; the two uses are completely different.

In the function call to `func_two`, parentheses force the compiler to interpret the first comma as the sequential-evaluation operator. This function call passes two arguments to `func_two`. The first argument is the result of the sequential-evaluation operation `(x--, y + 2)`, which has the value and type of the expression `y + 2`; the second argument is `z`.

5.3.12 Conditional Operator

C has one ternary operator, the conditional operator (? :). Its form is as follows:

operand1 ? *operand2* : *operand3*

The expression *operand1* is evaluated in terms of its equivalence to 0. It must have integral, floating-point, or pointer type. If *operand1* has a nonzero value, *operand2* is evaluated and the result of the expression is the value of *operand2*. If *operand1* evaluates to 0, *operand3* is evaluated, and the result of the expression is the value of *operand3*. Note that either *operand2* or *operand3* is evaluated, but not both.

The type of the result depends on the types of the second and third operands, as follows:

1. If the second and third operands have integral or floating-point type (their types can be different), the usual arithmetic conversions are performed. The type of the result is the type of the operands after conversion.
2. The second and third operands can have the same structure, union, or pointer type. The type of the result is the same structure, union, or pointer type.
3. One of the second or third operands can be a pointer and the other a constant expression with the value 0. The type of the result is the pointer type.

Example

```
j = (i < 0) ? (-i) : (i);
```

The above example assigns the absolute value of `i` to `j`. If `i` is less than 0, `-i` is assigned to `j`. If `i` is greater than or equal to 0, `i` is assigned to `j`.

5.4 Assignment Operators

C's assignment operators can both transform and assign values in a single operation. Using a compound-assignment operator to replace two separate operations can reduce code size and improve program efficiency. The assignment operators are as follows:

Operator	Operation Performed
++	Unary increment
--	Unary decrement
=	Simple assignment
*=	Multiplication assignment
/=	Division assignment
%=	Remainder assignment
+=	Addition assignment
-=	Subtraction assignment
<<=	Left-shift assignment
>>=	Right-shift assignment
&=	Bitwise-AND assignment
|=	Bitwise-inclusive-OR assignment
^=	Bitwise-exclusive-OR assignment

In assignment, the type of the right-hand value is converted to the type of the left-hand value. The specific path of the conversion depends on the two types and is outlined in detail in Section 5.7.

5.4.1 Lvalue Expressions

An assignment operation specifies that the value of the right-hand operand is to be assigned to the storage location named by the left-hand operand. Therefore, the left-hand operand of an assignment operation (or the single operand of a unary assignment expression) must be an expression referring to a memory location. Expressions that refer to memory locations are called "lvalue" expressions. A variable name is such an expression: the name of the variable denotes a storage location, while the value of the variable is the value residing at that location.

The following C expressions may be lvalue expressions:

- Identifiers of character, integer, floating-point, pointer, enumeration, structure, or union type
- Subscript ([ ]) expressions, except when a subscript expression evaluates to a pointer to an array or a pointer to a function
- Member-selection expressions (− > and .), if the selected member is one of the aforementioned expressions
- Unary-indirection (*) expressions, except when such expressions refer to arrays or functions
- Type casts to pointer types, as long as the size of the object does not change
- An lvalue expression in parentheses

5.4.2 Unary Increment and Decrement

The unary assignment operators (++ and − −) increment and decrement their operand, respectively. The operand must have integral, floating-point, or pointer type, and must be an lvalue expression.

Operands of integral or floating-point type are incremented or decremented by the integer value 1. The type of the result is the type of the operand. An operand of pointer type is incremented or decremented by the size of the object it addresses. An incremented pointer points to the next object; a decremented pointer points to the previous object.

An increment (++) or decrement (− −) operator can appear either before or after its operand. When the operator prefixes its operand, the operand is incremented or decremented and its new value is the result of the expression. When the operator postfixes its operand, the immediate result of the expression is the value of the operand *before* it is incremented or decremented. After that result is noted in context, the operand is incremented or decremented.

Examples

```
/********************* Example 1 *********************/

if (pos++ > 0)
        *ptt = *qtt;
```

```
/********************* Example 2 *********************/

if (line[--i] != '\n')
        return;
```

In the first example, the variable `pos` is compared to 0, then incremented.

In the second example, the variable `i` is decremented before it is used as a subscript to `line`.

5.4.3 Simple Assignment

The simple assignment operator (=) performs assignment. The right operand is assigned to the left operand; the conversion rules for assignment apply (see Section 5.7.1).

Example

```
double x;
int y;

x = y;
```

The value of `y` is converted to **double** type and assigned to `x`.

5.4.4 Compound Assignment

The compound assignment operators consist of the simple assignment operator combined with another binary operator. Compound assignment operators perform the operation specified by the additional operator, then assign the result to the left operand. A compound assignment expression such as

expression1 += *expression2*

can be understood as

expression1 = *expression1* + *expression2*

However, the compound assignment expression is not equivalent to the expanded version because the compound assignment expression evaluates *expression1* only once, while in the expanded version *expression1* is evaluated twice: in the addition operation and in the assignment operation.

Each compound assignment operator performs the conversions that the corresponding binary operator performs, and restricts the types of its operands accordingly. The result of a compound assignment operation has the value and type of the left operand.

Example

```
#define MASK     0xffff

n |= MASK;
```

In this example a bitwise-inclusive-OR operation is performed on `n` and `MASK`, and the result is assigned to `n`. The manifest constant `MASK` is defined with a # **define** preprocessor directive, discussed in Section 8.2.1.

5.5 Precedence and Order of Evaluation

The precedence and associativity of C operators affect the grouping and evaluation of operands in expressions. An operator's precedence is meaningful only in the presence of other operators having higher or lower precedence. Expressions with higher-precedence operators are evaluated first.

Table 5.1 summarizes the precedence and associativity of C operators, listing them in order of precedence from highest to lowest. Where several operators appear together in a line or large brace, they have equal precedence and are evaluated according to their associativity.

Table 5.1

Precedence and Associativity of C Operators

Symbol[a]	Type of Operation	Associativity
( ) [ ] . − >	Expression	Left to right
− ~ ! * & ++ −− **sizeof** *casts*	Unary[b]	Right to left
* / %	Multiplicative	Left to right
+ −	Additive	Left to right
<< >>	Shift	Left to right
< > <= >=	Relational (inequality)	Left to right
== !=	Relational (equality)	Left to right
&	Bitwise-AND	Left to right
^	Bitwise-exclusive-OR	Left to right
|	Bitwise-inclusive-OR	Left to right
&&	Logical AND	Left to right
||	Logical OR	Left to right
? :	Conditional	Right to left
= *= /= %= += −= <<= >>= &= |= ^=	Simple and compound assignment[c]	Right to left
,	Sequential evaluation	Left to right

[a] Operators are listed in descending order of precedence. Where several operators appear in the same line or in a large brace, they have equal precedence.

[b] All unary operators have equal precedence.

[c] All simple and compound assignment operators have equal precedence.

As Table 5.1 shows, operands consisting of a constant, an identifier, a string, a function call, a subscript expression, a member-selection expression, or a parenthetical expression have highest precedence and associate left to right. Type-cast conversions have the same precedence and associativity as the unary operators.

An expression can contain several operators with equal precedence. When several such operators appear at the same level in an expression, evaluation proceeds according to the associativity of the operator, either right to left or left to right. The result of expressions involving multiple occurrences of multiplication (*), addition (+), or binary bitwise (& | ^) operators at the same level is indifferent to the direction of evaluation. The compiler is free to evaluate such expressions in any order, even when parentheses in the expression appear to specify a particular order.

Important

Only the sequential-evaluation operator (,) and the logical-AND (&&) and logical-OR (||) operators guarantee a particular order of evaluation for the operands. The sequential-evaluation operator (,) is guaranteed to evaluate its operands from left to right. (Note that the comma separating arguments in a function call is not the same as the sequential-evaluation operator and does not provide any such guarantee.)

The logical operators also guarantee evaluation of their operands left to right. However, the logical operators evaluate the minimum number of operands necessary to determine the result of the expression. Thus, some operands of the expression `x && y++`, the second operand, `y++`, is evaluated only if `x` is true (nonzero). Thus `y` is not incremented when `x` is false (0).

The examples below show the default groupings for several expressions:

Expression	Default Grouping
`a & b || c`	`(a & b) || c`
`a = b || c`	`a = (b || c)`
`q && r || s--`	`(q && r) || s--`

In the first example, the bitwise-AND operator (**&**) has higher precedence than the logical-OR operator (||), so `a & b` forms the first operand of the logical-OR operation.

In the second example, the logical-OR operator (||) has higher precedence than the simple assignment operator (=), so `b || c` is grouped as the right-hand operand in the assignment. Note that the value assigned to `a` is either 0 or 1.

The third example shows a correctly formed expression that may produce an unexpected result. The logical-AND operator (**&&**) has higher precedence than the logical-OR operator (||), so `q && r` is grouped as an operand. Since the logical operators guarantee evaluation of operands from left to right, `q && r` is evaluated before `s--`. However, if `q && r` evaluates to a nonzero value, `s--` is not evaluated, and `s` is not decremented. To correct this problem, `s--` should appear as the first operand of the expression or should be decremented in a separate operation.

The following example shows an illegal expression that produces a program error:

Illegal Expression	Default Grouping
`p == 0 ? p += 1: p += 2`	`(p == 0 ? p += 1 : p) += 2`

In this example, the equality operator (==) has the highest precedence, so `p == 0` is grouped as an operand. The ternary operator (**? :**) has the next-highest precedence. Its first operand is `p == 0` and its second operand is `p += 1`. However, the last operand of the ternary operator is considered to be `p` rather than `p += 2`, since this occurrence of `p` binds more closely to the ternary operator than it does to the compound assignment operator. A syntax error occurs because `+= 2` does not have a left-hand operand.

To prevent errors of this kind, and to produce more readable code, the use of parentheses is recommended. The preceding example can be corrected and clarified through the use of parentheses, as shown below:

```
(p == 0) ? (p += 1) : (p += 2)
```

5.6 Side Effects

"Side effects" are changes in the state of the machine that occur as a result of evaluating an expression. They occur whenever the value of a variable is changed. Any assignment operation has side effects, and any call to a function that contains assignment operations has side effects.

The order of evaluation of side effects is implementation dependent, except where the compiler guarantees a particular order of evaluation, as outlined in Section 5.5.

For example, side effects occur in the following function call:

```
add (i + 1, i = j + 2)
```

The arguments of a function call can be evaluated in any order. The expression `i + 1` may be evaluated before `i = j + 2`, or vice versa, with different results in each case.

Unary increment and decrement operations involve assignment and can cause side effects, as shown in the following example:

```
d = 0;
a = b++ = c++ = d++;
```

The value of `a` is unpredictable. The value of `d` (initially 0) could be assigned to `c`, then to `b`, and then to `a` before any of the variables are incremented. In this case `a` would be equal to 0.

A second method of evaluating this expression begins by evaluating the operand `c++ = d++`. The value of `d` (initially 0) is assigned to `c`, and then both `d` and `c` are incremented. Next, the value of `c`, which is now 1, is assigned to `b` and `b` is incremented. Finally, the incremented value of `b` is assigned to `a`; in this case, the final value of `a` is 2.

Since the C language does not define the order of evaluation of side effects, both of these evaluation methods are correct and either can be implemented. Statements that depend on a particular order of evaluation for side effects produce nonportable and unclear code.

5.7 Type Conversions

Type conversions occur when a value is assigned to a variable of a different type, when a value is explicitly cast to another type, when an operator converts the type of its operand or operands before performing an operation, and when a value is passed as an argument to a function. The rules governing each kind of conversion are outlined next.

5.7.1 Assignment Conversions

In assignment operations, the type of the value being assigned is converted to the type of the variable receiving the assignment. C allows conversions by assignment between integral and floating-point types, even when the conversion entails loss of information. The methods of carrying out the conversions depend upon the type, as follows.

5.7.1.1 Conversions from Signed Integral Types

A signed integer is converted to a shorter signed integer by truncating the high-order bits, and is converted to a longer signed integer by sign extension. Conversion of signed integers to floating-point values takes place without loss of information, except that some precision can be lost when a **long** value is converted to a **float**. In converting a signed integer to an unsigned integer, the signed integer is converted to the size of the unsigned integer and the result is interpreted as an unsigned value.

Conversions from signed integral types are summarized in Table 5.2. This table assumes that the **char** type is signed by default. If a compile-time option is used to change the default for the **char** type to unsigned, the conversions for the **unsigned char** type given in Table 5.3 apply.

Table 5.2
Conversions from Signed Integral Types

From	To	Method
char[a]	**short**	Sign extend
char	**long**	Sign extend
char	**unsigned char**	Preserve pattern; high-order bit loses function as sign bit
char	**unsigned short**	Sign extend to **short**; convert **short** to **unsigned short**
char	**unsigned long**	Sign extend to **long**; convert **long** to **unsigned long**
char	**float**	Sign extend to **long**; convert **long** to **float**
char	**double**	Sign extend to **long**; convert **long** to **double**
short	**char**	Preserve low-order byte
short	**long**	Sign extend
short	**unsigned char**	Preserve low-order byte
short	**unsigned short**	Preserve bit pattern; high-order bit loses function as sign bit
short	**unsigned long**	Sign extend to **long**; convert **long** to **unsigned long**
short	**float**	Sign extend to **long**; convert **long** to **float**
short	**double**	Sign extend to **long**; convert **long** to **double**
long	**char**	Preserve low-order byte
long	**short**	Preserve low-order word
long	**unsigned char**	Preserve low-order byte
long	**unsigned short**	Preserve low-order word
long	**unsigned long**	Preserve bit pattern; high-order bit loses function as sign bit
long	**float**	Represent as **float**; if **long** cannot be represented exactly, some loss of precision occurs
long	**double**	Represent as **double**; if **long** cannot be represented exactly as a **double**, some loss of precision occurs

[a] All **char** entries assume that the **char** type is signed by default.

Note

The **int** type is equivalent either to the **short** type or to the **long** type, depending on the implementation. Conversion of an **int** value proceeds as for a **short** or a **long**, whichever is appropriate.

5.7.1.2 Conversions from Unsigned Integral Types

An unsigned integer is converted to a shorter unsigned or signed integer by truncating the high-order bits. An unsigned integer is converted to a longer unsigned or signed integer by zero extending. Unsigned values are converted to floating-point values by converting first to a signed integer of the same size, then converting that signed value to a floating-point value.

When an unsigned integer is converted to a signed integer of the same size, no change in the bit pattern occurs. However, the value it represents changes if the sign bit is set.

Conversions from unsigned integral types are summarized in Table 5.3.

Table 5.3

Conversions from Unsigned Integral Types

From	To	Method
unsigned char	**char**	Preserve bit pattern; high-order bit becomes sign bit
unsigned char	**short**	Zero extend
unsigned char	**long**	Zero extend
unsigned char	**unsigned short**	Zero extend
unsigned char	**unsigned long**	Zero extend
unsigned char	**float**	Convert to **long**; convert **long** to **float**

Table 5.3 *(continued)*

From	To	Method
unsigned char	**double**	Convert to **long**; convert **long** to **double**
unsigned short	**char**	Preserve low-order byte
unsigned short	**short**	Preserve bit pattern; high-order bit becomes sign bit
unsigned short	**long**	Zero extend
unsigned short	**unsigned char**	Preserve low-order byte
unsigned short	**unsigned long**	Zero extend
unsigned short	**float**	Convert to **long**; convert **long** to **float**
unsigned short	**double**	Convert to **long**; convert **long** to **double**
unsigned long	**char**	Preserve low-order byte
unsigned long	**short**	Preserve low-order word
unsigned long	**long**	Preserve bit pattern; high-order bit becomes sign bit
unsigned long	**unsigned char**	Preserve low-order byte
unsigned long	**unsigned short**	Preserve low-order word
unsigned long	**float**	Convert to **long**; convert **long** to **float**
unsigned long	**double**	Convert to **long**; convert **long** to **double**

Note

The **unsigned int** type is equivalent either to the **unsigned short** type or to the **unsigned long** type, depending on the implementation. Conversion of an **unsigned int** value proceeds as for an **unsigned short** or an **unsigned long**, whichever is appropriate.

5.7.1.3 Conversions from Floating-Point Types

A **float** value converted to a **double** undergoes no change in value. A **double** converted to a **float** is represented exactly, if possible. If the value is too large to fit into a **float**, precision is lost.

A floating-point value is converted to an integer value by converting to a **long**. Conversions to other integer types occur as for a **long**. The decimal portion of the floating-point value is discarded in the conversion to a **long**; if the result is still too large to fit into a **long**, the result of the conversion is undefined.

Conversions from floating-point types are summarized in Table 5.4.

Table 5.4
Conversions from Floating-Point Types

From	To	Method
float	**char**	Convert to **long**; convert **long** to **char**
float	**short**	Convert to **long**; convert **long** to **short**
float	**long**	Truncate at decimal point; if result is too large to be represented as **long**, result is undefined
float	**unsigned short**	Convert to **long**; convert **long** to **unsigned short**
float	**unsigned long**	Convert to **long**; convert **long** to **unsigned long**
float	**double**	Change internal representation
double	**char**	Convert to **float**; convert **float** to **char**
double	**short**	Convert to **float**; convert **float** to **short**
double	**long**	Truncate at decimal point; if result is too large to be represented as **long**, result is undefined
double	**unsigned short**	Convert to **long**; convert **long** to **unsigned short**
double	**unsigned long**	Convert to **long**; convert **long** to **unsigned long**
double	**float**	Represent as a **float**. If **double** value cannot be represented exactly as **float**, loss of precision occurs; if value is too large to be represented as **float**, the result is undefined

5.7.1.4 Conversions to and from Pointer Types

A pointer to one type of value can be converted to a pointer to a different type. The result may be undefined, however, because of the alignment requirements and sizes of different types in storage.

In some implementations, the special keywords **near**, **far**, and **huge** are available to modify the size of pointers within a program. A pointer can be converted to a pointer of a different size; the path of the conversion is implementation dependent. For example, on an 8086 processor, the compiler might use a segment-register value to convert a 16-bit pointer to a 32-bit pointer. See your system documentation for information on pointer conversions.

A pointer value can be converted to an integral value. The path of the conversion depends on the size of the pointer and the size of the integral type, as follows:

- If the pointer is the same size as or larger than the integral type, the pointer behaves like an unsigned value in the conversion, except that it cannot be converted to a floating-point value.
- If the pointer is smaller than the integral type, the pointer is first converted to a pointer with the same size as the integral type, then converted to the integral type. The method of converting a pointer to a longer pointer is implementation dependent; see your system documentation for information on pointer conversions.

An integral type can be converted to a pointer type. If the integral type is the same size as the pointer type, the conversion simply causes the integral value to be treated as a pointer (an unsigned integer). If the size of the integral type is different from the size of the pointer type, the integral type is first converted to the size of the pointer, using the conversion paths given in Tables 5.2 and 5.3. It is then treated as a pointer value.

If the special keywords **near**, **far**, and **huge** are implemented, implicit conversions may be made on pointer values. In particular, the compiler may make assumptions about the default size of pointers and convert passed pointer values accordingly, unless a forward declaration is present to override the implicit conversion. See your system documentation for information on pointer conversions.

5.7.1.5 Conversions from Other Types

An **enum** value is an **int** value, by definition of the **enum** type. Conversions to and from an **enum** value proceed as for the **int** type. An **int** is equivalent to either a **short** or a **long**, depending on the implementation.

No conversions between structure or union types are allowed.

The **void** type has no value, by definition. Therefore, it cannot be converted to any other type, nor can any value be converted to **void** by assignment. A value can be explicitly cast to **void**, however, as discussed in Section 5.7.2.

5.7.2 Type-Cast Conversions

Explicit type conversions can be made by means of a type cast. A type cast has the form

(*type-name*)*operand*

where *type-name* specifies a particular type and *operand* is a value to be converted to the specified type. (Type names are discussed in Section 4.9.)

The conversion of *operand* occurs as though it had been assigned to a variable of the named type. The conversion rules for assignments (outlined in Section 5.7.1) apply to type casts as well. The type name **void** can be used in a cast operation, but the resulting expression cannot be assigned to any item.

5.7.3 Operator Conversions

The conversions performed by C operators depend on the operator and on the type of the operand or operands. Many operators perform the "usual arithmetic conversions," which are outlined in Section 5.3.1.

C permits some arithmetic with pointers. In pointer arithmetic, integer values are converted to express memory positions; see the discussions of additive operators (Section 5.3.6) and subscript expressions (Section 5.2.5) for information.

5.7.4 Function-Call Conversions

The type of conversion performed on the arguments in a function call depends on whether a forward declaration with declared argument types is present for the called function.

If a forward declaration is present, and it includes declared argument types, the compiler performs type checking. The type-checking process is outlined in detail in Section 7.4.1, "Actual Arguments."

If no forward declaration is present, or if the forward declaration omits the argument-type list, the only conversions performed on the arguments in the function call are the usual arithmetic conversions. These conversions are performed independently on each argument in the call. This means that a **float** value is converted to a **double**; a **char** or **short** value is converted to an **int**; and an **unsigned char** or **unsigned short** is converted to an **unsigned int**.

If the special keywords **near**, **far**, and **huge** are implemented, implicit conversions may also be made on pointer values passed to functions. These implicit conversions can be overridden by providing argument-type lists to allow the compiler to perform type checking. See your system documentation for information on pointer conversions.

Chapter 6
Statements

6.1 Introduction 135
6.2 The break Statement 137
6.3 The Compound Statement 138
6.4 The continue Statement 140
6.5 The do Statement 141
6.6 The Expression Statement 142
6.7 The for Statement 143
6.8 The goto and Labeled Statements 145
6.9 The if Statement 146
6.10 The Null Statement 148
6.11 The return Statement 149
6.12 The switch Statement 151
6.13 The while Statement 154

6.1 Introduction

The statements of a C program control the flow of program execution. In C, as in other programming languages, several kinds of statements are available to perform loops, to select other statements to be executed, and to transfer control. This chapter describes C statements in alphabetical order, as follows:

break statement

compound statement

continue statement

do statement

expression statement

for statement

goto statement

if statement

null statement

return statement

switch statement

while statement

C statements consist of keywords, expressions, and other statements. The following keywords appear in C statements:

break	**default**	**for**	**return**
case	**do**	**goto**	**switch**
continue	**else**	**if**	**while**

The expressions in C statements are the expressions discussed in Chapter 5, "Expressions and Assignments." Statements appearing within C statements may be any of the statements discussed in this chapter. A statement that forms a component of another statement is called the "body" of the enclosing statement. Frequently the statement body is a "compound" statement; that is, a single statement composed of one or more statements.

The compound statement is delimited by braces; all other C statements end with a semicolon.

Any C statement may be prefixed with an identifying label consisting of a name and a colon. Statement labels are recognized only by the **goto** statement and are therefore discussed in Section 6.8, "The goto and Labeled Statements."

When a C program is executed, its effect is the same as execution of the statements in order of their appearance in the program, except where a statement explicitly transfers control to another location.

6.2 The break Statement

Syntax

break;

Execution

The **break** statement terminates the execution of the smallest enclosing **do**, **for**, **switch**, or **while** statement in which it appears. Control passes to the statement following the terminated statement. A **break** statement appearing outside any **do**, **for**, **switch**, or **while** statement causes an error.

Within nested statements, the **break** statement terminates only the **do**, **for**, **switch**, or **while** statement immediately enclosing it. To transfer control out of the nested structure, a **return** or **goto** statement can be used.

Example

```
for (i = 0; i < LENGTH - 1; i++) {
        for (j = 0; j < WIDTH - 1; j++) {
                if (lines[i][j] == '\0') {
                        lengths[i] = j;
                        break;
                }
        }
}
```

The above example processes an array of variable-length strings stored in `lines`. The **break** statement causes an exit from the interior **for** loop after the terminating null character (**\0**) of each string is found and stored in `lengths [i]`. Control then returns to the outer **for** loop. The variable `i` is incremented and the process is repeated until `i` is greater than or equal to LENGTH.

6.3 The Compound Statement

Syntax

{
[[*declaration*]]
.
.
.
statement
[[*statement*]]
.
.
.
}

Execution

The effect of a compound statement's execution is that of the execution of its statements in order of their appearance, except where a statement explicitly transfers control to another location. The form and meaning of the declarations that can appear at the head of a compound statement are described in Chapter 4, "Declarations."

Example

```
if (i > 0) {
        line[i] = x;
        x++;
        i--;
}
```

A compound statement typically appears as the body of another statement, such as the **if** statement. In the above example, if `i` is greater than 0, all of the statements in the compound statement are executed in order.

Labeled Statements

Like other C statements, any of the statements in a compound statement can carry a label. Transfer into the compound statement by means of a **goto** is therefore possible. However, transferring into a compound statement is dangerous when the compound statement includes declarations that initialize variables. Declarations in a compound statement precede the executable statements, so transferring directly to an executable statement within the compound statement bypasses the initializations. The results are unpredictable.

6.4 The continue Statement

Syntax

continue;

Execution

The **continue** statement passes control to the next iteration of the **do**, **for**, or **while** statement in which it appears, bypassing any remaining statements in the **do**, **for**, or **while** statement body. Within a **do** or a **while** statement, the next iteration begins with the reevaluation of the **do** or **while** statement's expression. Within a **for** statement, the next iteration starts with the evaluation of the **for** statement's loop expression. It proceeds with the evaluation of the conditional expression and subsequent termination or reiteration of the statement body.

Example

```
while (i-- > 0) {
        x = f(i);
        if (x == 1)
                continue;
        y = x * x;
}
```

The statement body is executed if `i` is greater than 0. First `f(i)` is assigned to `x`; then, if `x` is equal to 1, the **continue** statement is executed. The rest of the statements in the body are ignored, and execution resumes at the top of the loop with the evaluation of `i-- > 0`.

6.5 The do Statement

Syntax

do
statement
while (*expression*);

Execution

The body of a **do** statement is executed one or more times until *expression* becomes false (0). First, the statement body is executed; then *expression* is evaluated. If *expression* is false, the **do** statement terminates and control passes to the next statement in the program. If *expression* is true (non-zero), the statement body is executed again, and *expression* is tested again. The statement body is executed repeatedly until *expression* becomes false.

The **do** statement may also terminate with the execution of a **break**, **goto**, or **return** statement within the statement body.

Example

```
do {
        y = f(x);
        x--;
} while (x > 0);
```

The two statements `y = f(x);` and `x--;` are executed, regardless of the initial value of `x`. Then `x > 0` is evaluated. If `x` is greater than 0, the statement body is executed again and `x > 0` is reevaluated. The statement body is executed repeatedly as long as `x` remains greater than 0. Execution of the **do** statement terminates when `x` becomes 0 or negative.

6.6 The Expression Statement

Syntax

expression;

Execution

The expression is evaluated, according to the rules outlined in Chapter 5, "Expressions and Assignments."

Examples

```
x = (y + 3);                    /* Example 1 */

x++;                            /* Example 2 */

f(x);                           /* Example 3 */
```

In C, assignments are expressions; the value of the expression is the value being assigned (sometimes called the "right-hand value"). In the first example, `x` is assigned the value of `y + 3`. In the second example, `x` is incremented.

The third example shows a function-call expression. The value of the expression is the value, if any, returned by the function. If a function returns a value, the expression statement usually incorporates an assignment to store the returned value when the function is called. If the return value is not assigned, as in the example, the function call is executed but the return value, if any, is not used.

6.7 The for Statement

Syntax

for ([[*init-expression*]]; [[*cond-expression*]]; [[*loop-expression*]])
statement

Execution

The body of a **for** statement is executed zero or more times until the optional *cond-expression* becomes false. The *init-expression* and *loop-expression* are optional expressions that can be used to initialize and modify values during the **for** statement's execution.

The first step in the execution of the **for** statement is the evaluation of *init-expression*, if present. Next, *cond-expression* is evaluated, with three possible results:

1. If the conditional expression is true (nonzero), the statement body is executed; then *loop-expression*, if present, is evaluated. The process then begins again with the evaluation of *cond-expression*.
2. If the conditional expression is omitted, the conditional expression is considered true, and execution proceeds exactly as described above. A **for** statement lacking *cond-expression* terminates only upon the execution of a **break, goto,** or **return** statement within the statement body.
3. If the conditional expression is false, execution of the **for** statement terminates and control passes to the next statement in the program.

A **for** statement may also terminate with the execution of a **break, return,** or **goto** statement within the statement body.

Example

```
for (i = space = tab = 0; i < MAX; i++) {
        if (line[i] == '\x20')
                space++;
        if (line[i] == '\t') {
                tab++;
                line[i] ='\x20';
        }
}
```

The above example counts space (`'\x20'`) and tab (`'\t'`) characters in the array of characters named `line` and replaces each tab character with a space. First `i`, `space`, and `tab` are initialized to 0. Then `i` is compared to the constant `MAX`; if `i` is less than `MAX`, the statement body is executed. Depending on the value of `line[i]`, the body of one or neither of the **if** statements is executed. Then `i` is incremented and tested against `MAX`; the statement body is executed repeatedly as long as `i` is less than `MAX`.

6.8 The goto and Labeled Statements

Syntax

goto *name*;
.
.
.
name: *statement*

Execution

The **goto** statement transfers control directly to the statement specified by *name*. The labeled statement is executed immediately after the **goto** statement is executed. An error results if no statement with the given label resides in the same function or if an identical label appears before more than one statement in the same function.

A statement label is meaningful only to a **goto** statement; when a labeled statement is encountered in any other context, the statement is executed without regard to the label.

Example

```
if (errorcode > 0)
        goto exit;
.
.
.
exit:
        return (errorcode);
```

In the example, a **goto** statement transfers control to the point labeled `exit` when an error occurs.

Forming Labels

A label name is simply an identifier, formed by following the same rules that govern the construction of identifiers (see Section 2.4). Each statement label must be distinct from other statement labels and identifiers in the same function.

6.9 The if Statement

Syntax

if (*expression*)
        statement1
[**else**
        statement2]

Execution

The body of an **if** statement is executed selectively, depending on the value of *expression*. First, *expression* is evaluated. If *expression* is true (nonzero), the statement immediately following it is executed. If *expression* is false, the statement following the **else** keyword is executed. If *expression* is false and the **else** clause is omitted, the statement following *expression* is ignored. Control then passes from the **if** statement to the next statement in the program.

Example

```
if (i > 0)
        y = x/i;
else {
        x = i;
        y = f(x);
}
```

In the example, the statement `y = x/i;` is executed if `i` is greater than 0. If `i` is less than or equal to 0, `i` is assigned to `x` and `f(x)` is assigned to `y`. Note that the statement forming the **if** clause ends with a semicolon.

Nesting

C does not offer an "else if" statement, but the same effect is achieved by nesting **if** statements. An **if** statement may be nested in either the **if** clause or the **else** clause of another **if** statement.

When nesting **if** statements and **else** clauses, use braces to group the statements and clauses into compound statements that clarify your intent. In the absence of braces, the compiler resolves ambiguities by pairing each **else** with the most recent **if** lacking an **else**.

Examples

```
/******************** Example 1 ********************/

if (i > 0)              /* Without braces */
        if (j > i)
                x = j;
        else
                x = i;


/******************** Example 2 ********************/

if (i > 0) {            /* With braces */
        if (j > i)
                x = j;
}
else
        x = i;
```

In the first example, the **else** is associated with the inner **if** statement. If `i` is less than or equal to 0, no value is assigned to `x`.

In the second version, the braces surrounding the inner **if** statement make the **else** clause part of the outer **if** statement. If `i` is less than or equal to 0, `i` is assigned to `x`.

6.10 The Null Statement

Syntax

;

Execution

A null statement is a statement containing only a semicolon; it may appear wherever a statement is expected. Nothing happens when a null statement is executed.

Example

```
for (i = 0; i < 10; line[i++] = 0)
        ;
```

Statements such as **do, for, if,** and **while** require that an executable statement appear as the statement body. The null statement satisfies the syntax requirement in cases that do not need a substantive statement body. In the above example, the third expression of the **for** statement initializes the first 10 elements of `line` to 0. The statement body is a null statement, since no further statements are necessary.

Labeling a Null Statement

The null statement, like any other C statement, may be prefixed by an identifying label. To label an item that is not a statement, such as the closing brace of a compound statement, you can insert and label a null statement immediately preceding the item to get the same effect.

6.11 The return Statement

Syntax

return [[*expression*]];

Execution

The **return** statement terminates the execution of the function in which it appears and returns control to the calling function. Execution resumes in the calling function at the point immediately following the call. The value of *expression*, if present, is returned to the calling function. If *expression* is omitted, the return value of the function is undefined.

Example

```
main()
{
        void draw(int,int);
        long sq(int);
        .
        .
        .
        y = sq(x);
        draw(x, y);
        .
        .
        .
}

long sq(x)
int x;
{
        return (x * x);
}

void draw(x,y)
int x, y;
{
        .
        .
        .
        return;
}
```

The `main` function calls two functions, `sq` and `draw`. The `sq` function returns the value of `x * x` to `main`; the return value is assigned to `y`. The `draw` function is declared as a **void** function and does not return a value. An attempt to assign the return value of `draw` would cause an error.

By convention, parentheses enclose the *expression* of the **return** statement, as shown above. The language does not require the parentheses.

Omitting the Return Statement

If no **return** statement appears in a function definition, control automatically returns to the calling function after the last statement of the called function. The return value of the called function is undefined. If a return value is not required, the function should be declared to have **void** return type.

6.12 The switch Statement

Syntax

switch (*expression*) {
    [[*declaration*]]
    .
    .
    .
    [[**case** *constant-expression* **:**]]
    .
    .
    .
        [[*statement*]]
        .
        .
        .
    [[**default :**
        statement]]
    [[**case** *constant-expression* **:**]]
    .
    .
    .
        [[*statement*]]
        .
        .
        .
}

Execution

The **switch** statement transfers control to a statement within its body. The statement receiving control is the statement whose **case** *constant-expression* matches the value of the *expression* in parentheses. Execution of the statement body begins at the selected statement and proceeds through the end of the body or until a statement transfers control out of the body.

The **default** statement is executed if no **case** *constant-expression* is equal to the value of the **switch** *expression.* If the **default** statement is omitted, and no **case** match is found, none of the statements in the **switch** body is executed.

The **switch** expression is an integral value that must be the size of an **int** or shorter. It can also be an **enum** value. If the *expression* is shorter than an **int**, it is widened to an **int** value. Each **case** *constant-expression* is then cast to the type of the **switch** expression. The value of each **case** *constant-expression* must be unique within the statement body.

The **case** and **default** labels of the **switch** statement body are significant only in the initial test that determines the starting point for execution of the statement body. All statements appearing between the statement where execution starts and the end of the body are executed regardless of their labels, unless a statement transfers control out of the body entirely.

Declarations may appear at the head of the compound statement forming the **switch** body, but initializations included in the declarations are not performed. The effect of the **switch** statement is to transfer control directly to an executable statement within the body, bypassing the lines that contain initializations.

Examples

```
/********************* Example 1 *********************/

switch (c) {
          case 'A':
                  capa++;
          case 'a':
                  lettera++;
          default :
                  total++;
        }

/********************* Example 2 *********************/

switch (i) {
          case -1:
                  n++;
                  break;
          case 0 :
                  z++;
                  break;
          case 1 :
                  p++;
                  break;
        }
```

In the first example, all three statements of the **switch** body are executed if `c` is equal to `'A'`. Execution control is transferred to the first statement

(`capa++;`) and continues in order through the rest of the body. If `c` is equal to `'a'`, `lettera` and `total` are incremented. Only `total` is incremented if `c` is not equal to `'A'` or `'a'`.

In the second example, a **break** statement follows each statement of the **switch** body. The **break** statement forces an exit from the **switch** after one statement in the body is executed. If `i` is equal to −1, only `n` is incremented. The **break** following the statement `n++;` causes execution control to pass out of the **switch** body, bypassing the remaining statements. Similarly, if `i` is equal to 0, only `z` is incremented; if `i` is equal to 1, only `p` is incremented. The final **break** statement is not strictly necessary, since control will pass out of the body at the end of the compound statement, but it is included for consistency.

Multiple Labels

A statement may carry multiple **case** labels, as the following example shows:

```
case 'a' :
case 'b' :
case 'c' :
case 'd' :
case 'e' :
case 'f' :  hexcvt(c);
```

Although any statement within the body of the **switch** statement may be labeled, no statement is required to carry a label. Statements without labels may be intermingled freely with labeled statements. Keep in mind, however, that once the **switch** statement passes control to a statement within the body, all succeeding statements in the block are executed, regardless of their labels.

6.13 The while Statement

Syntax

while (*expression*)
statement

Execution

The body of a **while** statement is executed zero or more times until *expression* becomes false (0). First, *expression* is evaluated. If the *expression* is initially false, the body of the **while** statement is never executed, and control passes from the **while** statement to the next statement in the program. If *expression* is true (nonzero), the body of the statement is executed. Following each execution of the statement body, *expression* is reevaluated; the body is executed repeatedly as long as *expression* remains true.

The **while** statement may also terminate with the execution of a **break**, **goto**, or **return** within the statement body.

Example

```
while (i >= 0) {
        string1[i] = string2[i];
        i--;
}
```

The above example copies characters from `string2` to `string1`. If `i` is greater than or equal to 0, `string2[i]` is assigned to `string1[i]` and `i` is decremented. When `i` reaches or falls below 0, execution of the **while** statement terminates.

Chapter 7
Functions

7.1 Introduction 157
7.2 Function Definitions 157
7.2.1 Storage Class 158
7.2.2 Return Type 158
7.2.3 Formal Parameters 160
7.2.4 Function Body 164
7.3 Function Declarations 164
7.4 Function Calls 166
7.4.1 Actual Arguments 169
7.4.2 Calls with a Variable Number of Arguments 172
7.4.3 Recursive Calls 173

7.1 Introduction

A function is an independent collection of declarations and statements, usually designed to perform a specific task. C programs have at least one function, the main function, and they may have other functions. The sections of this chapter describe how to define, declare, and call C functions.

A function *definition* specifies the name of the function, its formal parameters, and the declarations and statements that define its action. The function definition can also give the return type of the function and its storage class.

A function *declaration* establishes the name, return type, and storage class of a function whose explicit definition is given at another point in the program. The number and types of arguments to the function can also be specified in the function declaration. This allows the compiler to compare the types of the actual arguments and the formal parameters of a function. Function declarations are optional for functions whose return type is **int**. To ensure correct behavior, functions with other return types must be declared before they are called.

A function *call* passes execution control from the calling function to the called function. The actual arguments, if any, are passed by value to the called function. Execution of a **return** statement in the called function returns control and possibly a value to the calling function.

7.2 Function Definitions

A function definition specifies the name, formal parameters, and body of a function. It may also define the function's return type and storage class. A function definition has the following form:

[[*sc-specifier*]][[*type-specifier*]] *declarator* ([[*parameter-list*]])
[[*parameter-declarations*]]
function-body

The *sc-specifier* gives the function's storage class, which must be either **static** or **extern**. The *type-specifier* and *declarator* together specify the

function's return type and name. The *parameter-list* is a list (possibly empty) of formal parameters to be used by the function. The *parameter-declarations* establish the types of the formal parameters. The *function-body* is a compound statement containing local variable declarations and statements. The following sections describe in detail the parts of the function definition.

7.2.1 Storage Class

The storage-class specifier in a function definition gives the function either **static** or **extern** storage class. A function with **static** storage class is visible only in the source file in which it is defined. All other functions, whether they are given **extern** storage class explicitly or implicitly, are visible throughout all the source files that constitute the program.

When the storage-class specifier is omitted from a function definition, the storage class defaults to **extern**. The **extern** storage-class specifier can be explicitly specified in the function definition, but it is not required.

The storage-class specifier is required in a function definition in only one case: when the function is declared elsewhere in the same source file with the **static** storage-class specifier.

The **static** storage-class specifier can also be used when defining a function previously declared in the same source file without a storage-class specifier. Normally, a function declared without a storage-class specifier defaults to the **extern** class. However, if the function definition explicitly specifies the **static** class, the function is given **static** class instead.

7.2.2 Return Type

The return type of a function defines the size and type of value returned by the function. The type declaration has the form

[[*type-specifier*]] *declarator*

where *type-specifier*, together with the *declarator*, defines the function's return type and name. If no *type-specifier* is given, the return type **int** is assumed.

The *type-specifier* can specify any fundamental, structure, or union type. The *declarator* consists of the function identifier, possibly modified to declare a pointer type. Functions cannot return arrays or functions, but they can return pointers to any type, including arrays and functions.

The return type given in the function definition must match the return type in declarations of the function elsewhere in the program. Functions with **int** return type do not have to be declared before they are called; functions with other return types cannot be called before they are either defined or declared.

A function's return value type is used only when the function returns a value, which occurs when a **return** statement containing an expression is executed. The expression is evaluated, converted to the return value type if necessary, and returned to the point of call. If no **return** statement is executed, or if the executed **return** statement does not contain an expression, the return value of the function is undefined. If the calling function expects a return value, the behavior of the program is also undefined.

Examples

```
/********************* Example 1 *********************/

        /* return type is int */
    static add (x, y)
    int x, y;
    {
        return (x+y);
    }

/********************** Example 2 **********************/

    typedef struct  {
        char name[20];
        int id;
        long class;
    } STUDENT;

        /* return type is STUDENT */
    STUDENT sortstu (a, b)
    STUDENT a, b;
{
    return ( (a.id < b.id) ? a : b );
}
```

```
/********************** Example 3 **********************/

      /* return type is char pointer */
    char *smallstr(s1, s2)
    char s1[], s2[];
    {
        int i;

        i=0;
        while ( s1[i] != '\0' && s2[i] != '\0' )
                        i++;
        if ( s1[i] == '\0' )
                return (s1);
        else
                return (s2);
    }
```

In the first example, the return type of `add` is **int** by default. The function has **static** storage class, which means it can be called only by functions in the same source file.

The second example defines the `STUDENT` type with a **typedef** declaration and defines the function `sortstu` to have `STUDENT` return type. The function selects and returns one of its two structure arguments.

The third example defines a function returning a pointer to an array of characters. The function takes two character arrays (strings) as arguments and returns a pointer to the shorter of the two strings. A pointer to an array points to the type of the array elements; thus, the return type of the function is a pointer to **char**.

7.2.3 Formal Parameters

Formal parameters are variables that receive values passed to a function by a function call. The formal parameters are declared in a parameter list at the beginning of the function declaration. The parameter list defines the names of the parameters and the order in which they take on values in the function call.

The parameter list consists of zero or more identifiers, separated by commas. The list must be enclosed in parentheses, even if no identifiers are given.

A comma followed by three periods (,...) can appear after the last identifier in the parameter list, indicating that the number of arguments to the function is variable. However, the function is expected to have at least as many parameters as there are identifiers before the last comma.

A parameter list can also consist of three periods (...) and no identifiers. This indicates that the number of parameters to the function is variable and may be zero.

Note

To maintain compatibility with previous versions, the compiler will also accept the comma character, without the trailing periods, at the end of the parameter list to indicate a variable number of arguments. A single comma can also be used instead of three periods to form the parameter list of a function taking zero or more arguments. Use of the comma is supported only for compatibility; using the three periods is recommended for new code.

Parameter declarations define the type and size of values stored in the formal parameters. These declarations have the same form as other variable declarations (see Section 4.4). A formal parameter can have any fundamental, structure, union, pointer, or array type.

A parameter can only have **auto** or **register** storage class. If no storage class is given, **auto** storage is assumed. If a formal parameter is named in the parameter list but is not declared, the parameter is assumed to have **int** type. Formal parameters can be declared in any order.

The identifiers of the formal parameters are used in the function body to refer to the values passed to the function. These identifiers cannot be used for variable declarations within the function body.

Only identifiers that appear in the parameter list can be declared as formal parameters. If the function has a variable number of arguments, the programmer is responsible for determining the number of arguments passed, and for retrieving additional arguments from the stack within the body of the function. See your system documentation for information on macros that can be used to do this in a portable way.

The type of each formal parameter should correspond to the type of the actual argument and to the type of the corresponding argument in the argument-type list for the function, if such a list is present. The compiler performs the usual arithmetic conversions independently on each formal parameter and on each actual argument, if necessary. After conversion, no formal parameter is shorter than an **int**, and no formal parameter has **float** type. This means, for example, that declaring a formal parameter as a **char** has the same effect as declaring it as an **int**.

If the **near**, **far**, and **huge** keywords are implemented, the compiler may also perform conversions on any pointer arguments to a function. The conversions performed depend on the default size of pointers in the program and the presence or absence of an argument-type list for the function. See your system documentation for specific information on pointer conversions.

The converted type of each formal parameter determines how the arguments placed on the stack by the function call are interpreted. A type mismatch between an actual and a formal parameter can cause the arguments on the stack to be misinterpreted. For example, if a 16-bit pointer is passed as an actual argument, then declared as a **long** formal parameter, the first 32 bits on the stack are interpreted as a **long** formal parameter. This error creates problems not only with the **long** formal parameter, but with any formal parameters that follow it. Errors of this kind can be detected through diligent use of argument-type lists in function declarations.

Example

```
struct student {
        char name[20];
        int id;
        long class;
        struct student *nextstu;
} student;

main()
{
        int match ( struct student *, char * );
        .
        .
        .
```

```
        if (match (student.nextstu, student.name) > 0) {
        .
        .
        .
        }
}

match ( r, n )
struct student *r;
char *n;
{
        int i = 0;

        while ( r->name[i] == n[i] )
                if ( r->name[i++] == '\0' )
                        return (r->id);
        return (0);
}
```

The example contains a structure type declaration, a forward declaration of the function `match`, a call to `match`, and the definition of the `match` function. Note that the same name, `student`, can be used without conflict both for the structure tag and for the structure variable name.

The `match` function is declared to have two arguments, the first a pointer to the `student` structure type and the second a pointer to a **char** type.

The two formal parameters of the `match` function are `r` and `n`. The parameter `r` is declared as a pointer to the `student` structure type; the parameter `n` is declared as a pointer to a **char** type.

The function is called with two arguments, both members of the `student` structure. Because there is a forward declaration of `match`, the compiler performs type checking between the actual arguments and the argument-type list and between the actual arguments and the formal parameters. Since the types match, no warnings or conversions are necessary.

Note that the array name given as the second argument in the call evaluates to a **char** pointer. The corresponding formal parameter is also declared as a **char** pointer, and is used in subscripted expressions as though it were an array identifier. Since an array identifier evaluates to a pointer expression, the effect of declaring the formal parameter as `char *n` is the same as declaring it `char n[ ]`.

Within the function, the local variable `i` is defined and used to monitor the current position in the array. The function returns the `id` structure member if the `name` member matches the array `n`; otherwise, it returns 0.

7.2.4 Function Body

The function body is simply a compound statement. The compound statement contains the statements that define the function's action and can also contain declarations of variables used by these statements. See Section 6.3 of Chapter 6, "Statements," for a discussion of compound statements.

All variables declared in the function body have **auto** storage type unless otherwise specified. When the function is called, storage space for the local variables is created and local initializations are performed. Execution control passes to the first statement in the compound statement and continues sequentially until a **return** statement or the end of the function body is encountered. Control then returns to the point of call.

A **return** statement containing an expression must be executed if the function is to return a value. The return value of a function is undefined if no **return** statement is executed or if the **return** statement does not include the optional expression.

7.3 Function Declarations

A function declaration defines the name, return type, and storage class of a given function, and may establish the type of some or all of the function's arguments. See Chapter 4, "Declarations," for a detailed description of the syntax of function declarations.

Functions can be declared implicitly or with forward declarations. The return type of a function declared either implicitly or with a forward declaration must agree with the return type specified in the function definition.

An implicit declaration occurs whenever a function is called without being previously defined or declared. The C compiler implicitly declares the called function to have **int** return type. By default, the function is declared to have **extern** storage class; the function definition can redefine the storage class to **static**, provided the function definition is given later in the same source file.

A forward declaration establishes the attributes of a function, allowing the declared function to be called before it is defined or allowing it to be called from another source file. If the storage-class specifier **static** is given in a forward declaration, the function has **static** class. The function definition must also specify the **static** class. If the storage-class specifier is **extern** or

is omitted, the function has **extern** class. However, the function definition can redefine the storage class as **static**, provided the function definition appears below the declaration in the same source file.

Forward declarations have several important uses. They establish the return type for functions that return any type of value but **int**. (Functions that return **int** values can also have forward declarations, but do not require them.) If a function with non-**int** return type is called before it is declared or defined, the results are unpredictable.

Forward declarations can be used to establish the types of arguments expected in a function call. The optional argument-type list of a forward declaration gives the type and number of arguments expected. (The number of arguments can vary.) The argument-type list is a list of type names corresponding to the expression list in the function call.

If no argument-type list is supplied, no type checking is performed. Type mismatches between actual arguments and formal parameters are silently accepted. Type checking is discussed further in Section 7.4.1, "Actual Arguments."

Forward declarations are also used to declare pointers to functions before the functions are defined.

Example

```
main()
{
        int a = 0, b = 1;
        float x = 2.0, y = 3.0;
        double realadd(double, double);

        a = intadd (a, b);
        x = realadd(x, y);
}

intadd(a, b)
int a, b;
{
        return (a + b);
}

double realadd(x, y)
double x, y;
{
        return (x + y);
}
```

In the example, the function `intadd` is implicitly declared to return an **int** value, since it is called before it is defined. The compiler does not check the types of the arguments in the call because no argument-type list is available.

The function `realadd` returns a **double** value instead of an **int**. The forward declaration of `realadd` in the `main` function allows the `realadd` function to be called before it is defined. Note that the definition of `realadd` matches the forward declaration by specifying the **double** return type.

The forward declaration of `realadd` also establishes the types of its two arguments; the actual arguments match the types given in the forward declaration and also match the types of the formal parameters.

7.4 Function Calls

A function call is an expression that passes control and zero or more actual arguments to a function. A function call has the form

expression([[*expression-list*]])

where *expression* evaluates to a function address and *expression-list* is a list of expressions (separated by commas) whose values, the actual arguments, are passed to the function. The *expression-list* can be empty.

When the function call is executed, the expressions in the function expression list are copied, converted as necessary, and then passed to formal parameters of the called function. The first expression in the list always corresponds to the first formal parameter of the function, the second expression corresponds to the second formal parameter, and so on through the end of the list. Since the called function works with copies of the actual arguments, any changes it makes to the arguments are not reflected in the original values from which the copies were made.

Execution control then passes to the first statement in the function. The execution of a **return** statement in the body of the function returns control and possibly a value to the calling function. If no **return** statement is executed, control returns to the caller after the last statement of the called function is executed, and the return value is undefined.

Important

The expressions in the function call's expression list can be evaluated in any order, so expressions with side effects have unpredictable results. The only guarantee the compiler makes is that all side effects in the expression list are evaluated before control passes to the called function.

The only requirement in calling a function is for the expression before the parentheses to evaluate to a function address. This means that a function can be called through any function-pointer expression. It may be helpful to remember that a function is called in the same manner in which it is declared. For instance, when declaring a function, the name of the function is given, followed by an argument-type list in parentheses. To call the function, only the name of the function is required, followed by an expression list in parentheses. The indirection operator (*) is not required to call the function because the name of the function evaluates to the function address.

The same principle applies when calling a function through a pointer. For example, suppose a function pointer is declared as follows:

```
int (*fpointer)(int, int);
```

The identifier `fpointer` is declared to point to a function taking two **int** arguments and returning an **int** value. A function call through `fpointer` might look like this:

```
(*fpointer)(3,4)
```

The indirection operator (*) is used to obtain the address of the function to which `fpointer` points. The function address is then used to call the function.

Examples

```
/********************* Example 1 *********************/

    double *realcomp(double, double);
    double a, b, *rp;
    .
    .
    .
    rp = realcomp(a, b);

/********************* Example 2 *********************/

    main ()
    {
        long lift(int), step(int), drop(int);
        void work (int, long (*)(int));
        int select, count;
        .
        .
        .
        select = 1;
        switch ( select ) {
                case 1: work(count, lift);
                        break;

                case 2: work(count, step);
                        break;

                case 3: work(count, drop);

                default:
                        break;
        }
}

void work ( n, func )
int n;
long (*func)(int);
{
    int i;
    long j;

    for (i = j = 0; i < n; i++)
            j += (*func)(i);
}
```

In the first example, the `realcomp` function is called in the statement `rp = realcomp(a, b);`. Two **double** arguments are passed to the `realcomp` function; the return value, a pointer to a **double**, is assigned to `rp`.

In the second example, the function call

```
work (count, lift);
```

in `main` passes an integer variable and the address of the function `lift` to the function `work`. Note that the function address is passed simply by giving the function identifier, since a function identifier evaluates to a pointer expression. To use a function identifier in this way, the function must be declared or defined before the identifier is used; otherwise, the identifier is not recognized. In this case, a forward declaration for `work` is given at the beginning of the `main` function.

The formal parameter `func` in `work` is declared to be a pointer to a function taking one **int** argument and returning a **long**. The parentheses around the parameter name are required; without them, the declaration would specify a function returning a pointer to a **long**.

The function `work` calls the selected function by using the following function call:

```
(*func)(i);
```

One argument, `i`, is passed to the called function.

7.4.1 Actual Arguments

An actual argument can be any value with fundamental, structure, union, or pointer type. Although arrays and functions cannot be passed as parameters, pointers to these items can be passed.

All actual arguments are passed by value. A copy of the actual argument is assigned to the corresponding formal parameter. The function uses this copy without affecting the variable from which it was originally derived.

Pointers provide a way to access a value by reference from a function. Since a pointer to a variable holds the address of the variable, the function can use this address to access the value of the variable. Pointer arguments allow a function to access arrays and functions, even though arrays and functions cannot be passed as arguments.

The expressions in a function call are evaluated and converted as follows:

- If an argument-type list is present, then for each actual argument in the function call, the usual arithmetic conversions are performed independently on the corresponding type in the argument-type list, and the actual argument is converted to that type. Next, the converted expression is compared with the type of the formal parameter that has the same place in the parameter list that the expression has in the expression list. (The formal parameters also undergo the usual arithmetic conversions before the comparison.) No conversions are performed, but the compiler produces warning messages as if the expressions were assigned to the formal parameters.
- If no argument-type list is present, or if there are more actual arguments than there are type names in the argument-type list, the usual arithmetic conversions are performed independently on each actual argument that lacks a corresponding type name.

If the **near**, **far**, and **huge** keywords are implemented, implementation-dependent conversions on pointer arguments may also be performed. See your system documentation for information on pointer conversions.

The number of expressions given in the expression list must match the number of formal parameters, unless the function's forward declaration and possibly its definition explicitly specify a variable number of arguments. In this case, the compiler checks as many arguments as there are type names in the argument-type list and converts them, if necessary, as described above.

If the argument-type list contains the special type name **void**, the compiler expects zero actual arguments in the function call and zero formal parameters. It produces a warning message if it finds otherwise.

The type of each formal parameter also undergoes the usual arithmetic conversions. The converted type of each formal parameter determines how the arguments on the stack are interpreted; if the type of the formal parameter does not match the type of the actual argument, the data on the stack can be misinterpreted.

Note

Type mismatches between actual and formal parameters can produce serious errors, particularly when the mismatches entail size differences. Keep in mind that these errors are not detected unless an argument-type list is given in the forward declaration of the function.

Example

```
main ()
{
        void swap (int *, int *);
        int x, y;
        .
        .
        .
        swap(&x, &y);
}

void swap (a, b)
int *a, *b;
{
        int t;

        t = *a;
        *a = *b;
        *b = t;
}
```

In the above example, the `swap` function is declared in `main` to have two arguments, both pointers to integers. The formal parameters `a` and `b` are also declared as pointers to integer variables. In the function call

```
swap (&x, &y)
```

the address of `x` is stored in `a` and the address of `y` is stored in `b`. Now two names, or "aliases," exist for the same location. References to `*a` and `*b` in `swap` are effectively references to `x` and `y` in `main`. The assignments within `swap` change the contents of `x` and `y`.

The compiler performs type checking on the arguments to `swap` because an argument-type list is present in the forward declaration of `swap`. The types of the actual arguments match both the argument-type list and the formal parameters.

7.4.2 Calls with a Variable Number of Arguments

To call a function with a variable number of arguments, the programmer simply gives any number of arguments in the function call. In the forward declaration of the function (if there is one), a variable number of arguments is specified by placing a comma followed by three periods (,...) at the end of the argument-type list (see Section 4.5 of Chapter 4, "Declarations"). One argument must be present in the function call for each type name specified in the argument-type list. If only the three periods (but no type names) are given, no arguments are required when calling the function.

Similarly, the parameter list in the function definition can end with a comma followed by three periods (,...) to indicate a variable number of arguments. If the parameter list contains only three periods (...), the number of parameters is variable and may be zero. See Section 7.2, "Function Definitions," for more information on the form of the parameter list.

Note

To maintain compatibility with previous versions, the compiler will also accept the comma character, without the trailing periods, at the end of the argument-type list or parameter list to indicate a variable number of arguments. A single comma can also be used instead of three periods to form the argument-type list or parameter list of a function taking zero or more arguments. Use of the comma is supported only for compatibility; use of the three periods is recommended for new code.

All the arguments given in the function call are placed on the stack. The number of formal parameters declared for the function determines how many of the arguments are taken from the stack and assigned to the formal parameters. The programmer is responsible for retrieving any additional arguments from the stack and for determining how many arguments are present. See your system documentation for information about macros that can be used to handle a variable number of arguments in a portable way.

7.4.3 Recursive Calls

Any function in a C program can be called recursively. A function can therefore call itself. The C compiler allows any number of recursive calls to a function. On each call, new storage is allocated for the formal parameters and for the **auto** and **register** variables so that their values in previous, unfinished calls are not overwritten. Previous parameters are inaccessible to all versions of the function except the version in which they were created.

Note that variables declared with global storage do not require new storage with each recursive call. Their storage exists for the lifetime of the program. Each reference to such a variable accesses the same storage area.

Although the C compiler defines no limit for the number of times a function can be called recursively, the operating environment may impose a practical limit. Since each recursive call requires additional stack memory, too many recursive calls can cause a stack overflow.

Chapter 8

Preprocessor Directives and Pragmas

8.1 Introduction 177

8.2 Manifest Constants and Macros 178

8.2.1 The #define Directive 178

8.2.2 The #undef Directive 182

8.3 Include Files 183

8.4 Conditional Compilation 184

8.4.1 The #if, #elif, #else, and #endif Directives 185

8.4.2 The #ifdef and #ifndef Directives 188

8.5 Line Control 189

8.6 Pragmas 190

8.1 Introduction

A "preprocessor directive" is an instruction intended for the C preprocessor. The C preprocessor is a text processor used to manipulate the text of a source file as the first phase of compilation. The compiler ordinarily invokes the preprocessor in its first pass, but the preprocessor can also be invoked separately to process text without compiling.

Preprocessor directives are typically used to make source programs easy to modify and to compile in different execution environments. Directives in the source file instruct the preprocessor to perform specific actions. For example, the preprocessor can replace tokens in the text, insert the contents of other files into the source file, and suppress compilation of a portion of the file by removing sections of text.

The C preprocessor recognizes the following directives:

#define	**#if**	**#line**
#elif	**#ifdef**	**#undef**
#else	**#ifndef**	
#endif	**#include**	

The number sign (#) must be the first non-white-space character on the line containing the directive; white-space characters can appear between the number sign and the first letter of the directive. Some directives are followed by arguments or values, as described below. Directives can appear anywhere in a source file, but they apply only to the remainder of the source file in which they appear.

A "pragma" is a "pragmatic," or practical, instruction to the C compiler. Pragmas are embedded in C source files and are typically used to control the actions of the compiler in a particular portion of a program without affecting the program as a whole. Section 8.6 describes the syntax for pragmas. However, the particular pragmas that are available, and their meanings, are defined by the implementation. See your system documentation for information on the use and effects of pragmas.

8.2 Manifest Constants and Macros

The #**define** directive is typically used to associate meaningful identifiers with constants, keywords, and commonly used statements or expressions. Identifiers that represent constants are called "manifest constants." Identifiers that represent statements or expressions are called "macros."

Once an identifier is defined, it cannot be redefined to a different value without first removing the definition. However, the identifier can be redefined with exactly the same definition. Thus, a program is allowed to contain more than one occurrence of the same definition.

The #**undef** directive removes the definition of an identifier. Once the definition has been removed, the identifier can be redefined to a different value. Sections 8.2.1 and 8.2.2 discuss the #**define** and #**undef** directives, respectively.

Macros can be defined to look and act like function calls. Because macros do not generate actual function calls, replacing function calls with macros can improve execution time. However, macros create problems if they are not defined and used with care. Macro definitions with arguments may require the use of parentheses to preserve the proper precedence in an expression. In addition, macros may not correctly handle expressions with side effects. See the examples in Section 8.2.1 for more information.

8.2.1 The #define Directive

Syntax

#**define** *identifier text*
#**define** *identifier*(*parameter-list*) *text*

The #**define** directive substitutes the given *text* for subsequent occurrences of the specified *identifier* in the source file. The *identifier* is replaced only when it forms a token. (Tokens are described in Chapter 2, "Elements of C," and in Appendix B, "Syntax Summary.") For instance, the *identifier* is not replaced when it occurs within a string or as part of a longer identifier.

If a *parameter-list* appears after the *identifier*, the #**define** directive replaces each occurrence of *identifier*(*parameter-list*) with a version of *text* modified by substituting actual arguments for formal parameters.

The *text* consists of a series of tokens, such as keywords, constants, or complete statements. One or more white-space characters must separate the *text* from the *identifier* (or from the closing parenthesis of the *parameter-list*). If the text is longer than one line, it can be continued onto the next line by preceding the new-line character with a backslash (\).

The *text* can also be empty. The effect of this option is to remove instances of the given *identifier* from the source file. The *identifier* is still considered defined, however, and yields the value 1 when tested with the #**if** directive (discussed in Section 8.4.1).

The *parameter-list*, when given, consists of one or more formal parameter names separated by commas. Each name in the list must be unique, and the list must be enclosed in parentheses. No spaces between the *identifier* and the opening parenthesis are allowed.

Formal parameter names appear in *text* to mark the places where actual values will be substituted. Each parameter name can occur more than once in the *text*, and the names can appear in any order.

The actual arguments following an instance of the *identifier* in the source file are matched to the formal parameters of the *parameter-list*, and the *text* is modified by replacing each formal parameter with the corresponding actual argument. The actual argument list and the formal *parameter-list* must have the same number of arguments.

Arguments with side effects sometimes cause macros to produce unexpected results. A macro definition may contain more than one occurrence of a given formal parameter, and if that formal parameter is replaced by an expression with side effects, the expression, with its side effects, is evaluated more than once (see Example 4 below).

Examples

```
/********************* Example 1 *********************/

#define WIDTH        80
#define LENGTH       (WIDTH + 10)

/********************* Example 2 *********************/

#define FILEMESSAGE "Attempt to create file \
failed because of insufficient space"
```

```
/******************** Example 3 ********************/

#define REG1          register
#define REG2          register
#define REG3

/******************** Example 4 ********************/

#define MAX(x,y)      ((x) > (y)) ? (x) : (y)

/******************** Example 5 ********************/

#define MULT(a,b)     ((a) * (b))
```

Example	Description

1	The first example defines the identifier `WIDTH` as the integer constant 80, and defines `LENGTH` in terms of `WIDTH` and the integer constant 10. Each occurrence of `LENGTH` is replaced with `( WIDTH + 10)`, which is in turn replaced with the expression `(80 + 10)`. The parentheses around `WIDTH + 10` are important because they control the interpretation in a statement such as the following:
	`var = LENGTH * 20;`
	After the preprocessing stage the statement becomes
	`var = (80 + 10) * 20;`
	or 1800. Without parentheses, the result is
	`var = 80 + 10 * 20;`
	which evaluates to 280 because the multiplication operator (*) has higher precedence than the addition operator (+).

2	The second example defines the identifier `FILEMESSAGE`. The definition is extended to a second line by using the backslash escape character (\).
3	The third example defines three identifiers, `REG1`, `REG2`, and `REG3`. `REG1` and `REG2` are defined as the keyword **register**. The definition of `REG3` is empty, so each occurrence of `REG3` is removed from the source file. These directives can be used to ensure that the program's most important

variables (declared with REG1 and REG2) are given **register** storage. See the discussion of the #**if** directive in Section 8.4.1 for an expanded version of this example.

4 The fourth example defines a macro named MAX. Each occurrence of the identifier MAX following the definition in the source file is replaced by the expression `((x) > (y)) ? (x) : (y)`, where actual values replace the parameters x and y. For example, the occurrence

```
MAX(1,2)
```

is replaced with

```
((1) > (2)) ? (1) : (2)
```

and the occurrence

```
MAX(i,s[i])
```

is replaced with

```
((i) > (s[i])) ? (i) : (s[i])
```

This macro is easier to read than the corresponding expression, making the source program easier to understand.

Note that arguments with side effects may cause this macro to produce unexpected results. For example, the occurrence `MAX(i, s[i++])` is replaced with `((i) > (s[i++])) ? (i) : (s[i++])`. The expression `(s[i++])` is evaluated twice, so by the time the ternary expression has been fully evaluated, i has increased by 2. The result of the ternary expression is unpredictable, since the operands of the ternary expression can be evaluated in any order, and the value of i varies depending on the evaluation order.

5 The fifth example defines the macro MULT. Once the macro is defined, an occurrence such as `MULT(3, 5)` is replaced by `(3) * (5)`. The parentheses around the parameters are important because they control the interpretation when complex expressions form the arguments to the macro.

For instance, the occurrence `MULT(3 + 4, 5 + 6)` is replaced by `(3 + 4) * (5 + 6)`, which evaluates to 77. Without the parentheses, the result is `3 + 4 * 5 + 6`, which evaluates to 29 because the multiplication operator (*) has higher precedence than the addition operator (+).

8.2.2 The #undef Directive

Syntax

undef *identifier*

The **#undef** directive removes the current definition of *identifier*. The preprocessor ignores subsequent occurrences of *identifier*. To remove a macro definition using **#undef**, give only the macro *identifier*; do not give a parameter list.

The **#undef** directive can also be applied to an identifier that has no previous definition. This ensures that the identifier is undefined.

The **#undef** directive is typically paired with a **#define** directive to create a region in a source program in which an identifier has a special meaning. For example, a specific function of the source program can use manifest constants to define environment-specific values that do not affect the rest of the program. The **#undef** directive also works with the **#if** directive (see Section 8.4.1) to control compilation of portions of the source program.

Example

```
#define WIDTH           80
#define ADD(X,Y)        (X) + (Y)
.
.
.
#undef WIDTH
#undef ADD
```

In this example, the **#undef** directive removes definitions of a manifest constant and a macro. Note that only the identifier of the macro is given.

8.3 Include Files

Syntax

include "*pathname*"
include <*pathname*>

The #**include** directive adds the contents of a given "include file" to another file. Constant and macro definitions can be organized into include files and added to any source file by using #**include** directives. Include files are also useful for incorporating declarations of external variables and complex data types. The types need only be defined and named once in an include file created for that purpose.

The #**include** directive tells the preprocessor to treat the contents of the named file as if they appeared in the source program at the point of the directive. The new text can also contain preprocessor directives. The preprocessor carries out directives in the new text, then continues processing the original text of the source file.

The *pathname* is a file name optionally preceded by a directory specification. It must name an existing file. The syntax of the file specification depends on the specific operating system on which the program is compiled.

The preprocessor uses the concept of a "standard" directory or directories to search for included files. The location of the standard directories for include files depends on the implementation and the operating system. See your system documentation for a definition of the standard directories.

The preprocessor stops searching as soon as it finds a file with the given name. If a complete, unambiguous path name for the include file is given, either in double quotation marks (" ") or in angle brackets (< >), the preprocessor searches only that path name and ignores the standard directories.

If the file specification does not give a complete path name, and the file specification is enclosed in double quotation marks, the preprocessor first searches for the file in the same directory as the including file (the "current working directory"). The preprocessor then searches directories specified in the compiler command line and finally searches the standard directories.

If the file specification is enclosed in angle brackets, the preprocessor does not search the current working directory. It begins by searching for the file in directories specified in the compiler command line and then searches the standard directories.

An #**include** directive can be nested; in other words, the directive can appear in a file named by another #**include** directive. When the preprocessor encounters the nested #**include** directive, it processes the named file and inserts it into the current file. The preprocessor uses the same search procedures outlined above in searching for nested include files.

The new file can also contain #**include** directives. Nesting can continue up to 10 levels. Once the nested #**include** is processed, the preprocessor continues to insert the enclosing include file into the original source file.

Examples

```
#include <stdio.h>                    /* Example 1 */

#include "defs.h"                     /* Example 2 */
```

The first example adds the contents of the file named `stdio.h` to the source program. The angle brackets cause the preprocessor to search the standard directories for `stdio.h`, after searching directories specified in the command line.

The second example adds the contents of the file specified by `defs.h` to the source program. The double quotation marks mean that the directory containing the current source file is searched first.

8.4 Conditional Compilation

This section describes the syntax and use of directives that control "conditional compilation." These directives allow for suppressing compilation of portions of a source file by testing a constant expression or identifier to determine which text blocks are passed on to the compiler and which are removed from the source file in the preprocessing stage.

8.4.1 The #if, #elif, #else, and #endif Directives

Syntax

```
# if restricted-constant-expression
    [[ text ]]
[[ # elif restricted-constant-expression
    text ]]
[[ # elif restricted-constant-expression
    text ]]
.
.
.
[[ # else
    text ]]
# endif
```

The #**if** directive, together with the #**elif**, #**else**, and #**endif** directives, controls compilation of portions of a source file. Each #**if** directive in a source file must be matched by a closing #**endif** directive. Zero or more #**elif** directives can appear between the #**if** and #**endif** directives, but at most one #**else** directive is allowed. The #**else** directive, if present, must be the last directive before #**endif.**

The preprocessor selects one of the given blocks of *text* for further processing. A *text* block is any sequence of text. It can occupy more than one line. Usually the *text* block is program text that has meaning to the compiler or the preprocessor. However, this is not a requirement; the preprocessor can be used to process any kind of text.

The selected *text* is processed by the preprocessor and passed to the compiler. If the *text* contains preprocessor directives, those directives are carried out.

Any text blocks not selected by the preprocessor are removed from the file in the preprocessing stage and are therefore not compiled.

The preprocessor selects a single *text* block by evaluating the *restricted-constant-expressions* following each #**if** or #**elif** directive until a true (nonzero) *restricted-constant-expression* is found. All *text* between the first true *restricted-constant-expression* and the next number sign (#) is selected.

If no *restricted-constant-expression* is true, or if there are no #**elif** directives, the preprocessor selects the text after the #**else** clause. If the #**else** clause is omitted, and no *restricted-constant-expression* in the #**if** block is true, no text is selected.

Each *restricted-constant-expression* follows the rules for restricted constant expressions discussed in Section 5.2.10 of Chapter 5, "Expressions and Assignments." Such expressions cannot contain **sizeof** expressions, type casts, or enumeration constants, but they can contain the special constant expression **defined**(*identifier*). This constant expression is considered true (nonzero) if the given *identifier* is currently defined; otherwise, the condition is false (0). An identifier defined as empty text is considered defined.

The #**if**, #**elif**, #**else**, and #**endif** directives can nest in the text portions of other #**if** directives. When nested, each #**else**, #**elif**, and #**endif** directive belongs to the closest preceding #**if** directive.

Examples

```
/********************* Example 1 *********************/

#if defined(CREDIT)
    credit();
#elif defined(DEBIT)
    debit();
#else
    printerror();
#endif

/********************* Example 2 *********************/

#if DLEVEL > 5
    #define SIGNAL  1
    #if STACKUSE == 1
                    #define STACK   200
    #else
                    #define STACK   100
    #endif
#else
    #define SIGNAL  0
    #if STACKUSE == 1
                    #define STACK   100
    #else
                    #define STACK   50
    #endif
#endif
```

```
/******************** Example 3 ********************/

#if DLEVEL == 0
    #define STACK 0
#elif DLEVEL == 1
    #define STACK 100
#elif DLEVEL > 5
    display( debugptr );
#else
    #define STACK 200
#endif

/******************** Example 4 ********************/

#define REG1    register
#define REG2    register

#if defined(M_86)
    #define REG3
    #define REG4
    #define REG5
#else
    #define REG3    register
    #if defined(M_68000)
            #define REG4    register
            #define REG5    register
    #endif
#endif
```

In the first example, the #**if** and #**endif** directives control compilation of one of three function calls. The function call to `credit` is compiled if the identifier `CREDIT` is defined. If the identifier `DEBIT` is defined, the function call to `debit` is compiled. If neither identifier is defined, the call to `printerror` is compiled. Note that `CREDIT` and `credit` are distinct identifiers in C because their cases are different.

The next two examples assume a previously defined manifest constant, `DLEVEL`. The second example shows two sets of nested #**if**, #**else**, and #**endif** directives. The first set of directives is processed only if `DLEVEL > 5` is true. Otherwise, the second set is processed.

In the third example, #**elif** and #**else** directives are used to make one of four choices, based on the value of `DLEVEL`. The manifest constant `STACK` is set to 0, 100, or 200, depending on the definition of `DLEVEL`. If `DLEVEL` is greater than 5, `display(debugptr);` is compiled and `STACK` is not defined.

The fourth example uses preprocessor directives to control the meaning of **register** declarations in a portable source file. The compiler assigns register storage to variables in the same order in which the **register** declarations appear in the source file. If a program contains more **register** declarations than the machine can accommodate, the compiler honors earlier declarations over later ones. Loss of efficiency can occur if the variables declared later are more heavily used.

The definitions listed above can be used to give priority to the most important register declarations. REG1 and REG2 are defined as the **register** keyword to declare **register** storage for the two most important variables in the program. For example, in the following fragment, b and c have higher priority than a or d.

```
func(a)

REG3 int a;

{
        REG1 int b;
        REG2 int c;
        REG4 int d;
        .
        .
        .
}
```

When M_86 is defined, the preprocessor removes the REG3 identifier from the file by replacing it with empty text; this prevents a from receiving **register** storage at the expense of b and c. When M_68000 is defined, all four variables are declared to have **register** storage. When neither M_86 nor M_68000 is defined, a, b, and c are declared with **register** storage.

8.4.2 The #ifdef and #ifndef Directives

Syntax

#ifdef *identifier*
#ifndef *identifier*

The **#ifdef** and **#ifndef** directives accomplish the same task as the **#if** directive used with **defined(***identifier***)**. These directives can be used anywhere **#if** can be used, and are provided only for compatibility with previous versions of the language. The **defined(***identifier***)** constant expression used with the **#if** directive is preferred.

When the preprocessor encounters an #**ifdef** directive, it checks to see whether the *identifier* is currently defined. If so, the condition is true (nonzero); otherwise, the condition is false (0).

The #**ifndef** directive checks for the opposite condition checked by #**ifdef**. If the identifier has not been defined (or its definition has been removed with #**undef**), the condition is true (nonzero). Otherwise, the condition is false (0).

8.5 Line Control

Syntax

line *constant* [["*filename*"]]

The #**line** directive instructs the preprocessor to change the compiler's internally stored line number and file name to a given line number and file name. The compiler uses the internally stored line number and file name to refer to errors encountered during compilation. The line number normally refers to the current input line; the file name refers to the current input file. The line number is increased after each line is processed.

Changing the line number and file name causes the compiler to ignore the previous values and to continue processing with the new values. The #**line** directive is typically used by program generators to cause error messages to refer to the original source file instead of the generated program.

The *constant* value in the #**line** directive is any integer constant. The *filename* can be any combination of characters and must be enclosed in double quotation marks (" "). If *filename* is omitted, the previous file name remains unchanged.

The current line number and file name are always available through the predefined identifiers __**LINE**__ and __**FILE**__. The __**LINE**__ and __**FILE**__ identifiers can be used to insert self-descriptive error messages into the program text.

The __**FILE**__ identifier contains a string representing the file name, surrounded by double quotation marks (""). Thus, it is not necessary to enclose the __**FILE**__ identifier in quotation marks when using it as a string.

Examples

```
/******************** Example 1 ********************/

#line 151 "copy.c"

/******************** Example 2 ********************/

#define ASSERT(cond)        if(!cond)\
{printf("assertion error line %d, file(%s)\n", \
__LINE__, __FILE__ );} else ;
```

In the first example, the internally stored line number is set to 151 and the file name is changed to `copy.c`.

In the second example, the macro `ASSERT` uses the predefined identifiers **__LINE__** and **__FILE__** to print an error message about the source file if a given "assertion" is not true. Note that no quotation marks are necessary around **__FILE__**.

8.6 Pragmas

Syntax

pragma *character-sequence*

A #**pragma** is an implementation-defined instruction to the compiler. It has the general form given above, where *character-sequence* is a series of characters giving a specific compiler instruction and arguments, if any. The number sign (#) must be the first non-white-space character on the line containing the pragma; white-space characters can appear between the number sign and the word **pragma**.

See your system documentation for information about the pragmas available in your compiler implementation.

Appendixes

Language Reference Appendixes

A Differences 193
B Syntax Summary 199

Appendix A
Differences

This appendix summarizes differences between Microsoft C and the description of the C language found in Appendix A of *The C Programming Language* by Brian W. Kernighan and Dennis M. Ritchie, published in 1978 by Prentice-Hall, Inc. The following is a list of the differences, with cross-references to the corresponding section numbers in *The C Programming Language*:

Section Number in Kernighan and Ritchie	Microsoft C
2.2	Identifiers (including those used in preprocessor directives) are significant to 31 characters. External identifiers are also significant to 31 characters.
2.3	The identifiers **asm** and **entry** are no longer keywords. New keywords are **const**, **volatile**, **enum**, **signed**, and **void**. (The **const** and **volatile** keywords are not yet implemented but are reserved for future use.) The identifiers **cdecl**, **far**, **fortran**, **huge**, **near**, and **pascal** may be keywords, depending on whether or not the corresponding options are enabled when a program is compiled (see your system documentation).
2.4.1	As a result of the method used for assigning types to hexadecimal and octal constants, these constants always act like unsigned integers in type conversions.

2.4.3

Hexadecimal bit patterns consisting of a backslash (\), the letter **x**, and up to two hexadecimal digits are permitted as character constants (for example, `\x12`).

Microsoft C defines three additional escape sequences: **v** represents a vertical tab (VT), \" represents the double-quote character, and **a** represents the bell (also called alert).

Character constants always have type **int**, with the result that they are sign extended in type conversions.

2.6

The **short** type is always 16 bits in length, the **long** type 32 bits. The size of an **int** is machine dependent. On 8086/8088, 80186, and 80286 processors an **int** is 16 bits long, and on 80386 and 68000 machines it is 32 bits.

4

The **char** type is signed by default, with the result that a **char** value is sign extended in type conversions. (In some implementations, the default for the **char** type can be changed to unsigned at compile time.)

Two additional unsigned types are supported: **unsigned char** and **unsigned long**.

The keyword **unsigned** or **signed** can be applied as an adjective to an integer type. When **unsigned** appears alone it means **unsigned int**; similarly, when **signed** appears alone, it means **int**.

Microsoft C offers an additional fundamental type, the **enum** (enumeration) type. The **void** type is defined as the return type of functions that do not return a value.

6.4 If the **near**, **far**, and **huge** keywords are enabled, pointers of different sizes may occur in a program. Operations with pointers of different sizes may cause conversion of pointers; the path of the conversion is implementation dependent.

6.6 The arithmetic conversions carried out by the Microsoft C Compiler are outlined in Sections 5.3.1 and 5.7 of Chapter 5, "Expressions and Assignments." Although compatible with the Kernighan and Ritchie conversions, the Microsoft C conversions are described in greater detail, including the specific path for each type of conversion.

In addition to the usual arithmetic conversions, conversions between pointers of different sizes may be routinely carried out when the **near**, **far**, and **huge** keywords are enabled. The path of the pointer conversions is implementation dependent.

7.2 In connection with the **sizeof** operator, a byte is defined as an 8-bit quantity.

7.14 A structure can be assigned to another structure of the same type.

8.2 The keywords **enum** and **void** are additional type specifiers. The keyword **signed** or **unsigned** can serve either as a type specifier or as an adjective modifying an integer type.

Therefore, the following additional combinations are acceptable:

signed char
signed short
signed short int
signed long
signed long int
unsigned char
unsigned short
unsigned short int
unsigned long
unsigned long int

8.4 Optional argument-type lists can be included in function declarations to notify the compiler of the number and types of arguments expected in a function call.

8.5 Bit fields must be declared **unsigned**.

The names of structure and union members are not required to be distinct from structure and union tags or from the names of other variables.

No relationship exists between the members of two different structure types.

8.6 Unions can be initialized by giving a value for the first member of the union.

9.7 The *expression* of a **switch** is an integral expression that is the size of an **int** or shorter. An *expression* with **enum** type is permitted. Each of the **case** constant expressions is cast to the type of the *expression*.

10.1 The parameter list in a function definition can end with a comma followed by three periods (,...) or just a comma (,) to indicate that the number of parameters is variable. A

parameter list containing only three periods (...) or a comma (,) indicates that the function can take zero or more parameters.

12 The number sign (#) introducing the preprocessor directive can be preceded by any combination of white-space characters. White space can also occur between the number sign and the preprocessor keyword.

In addition to preprocessor directives, the source file can also contain pragmas. Pragmas, like directives, are introduced by a number sign as the first non-white-space character in a line. The action defined by a particular pragma is implementation dependent.

12.3 The new combination #**if** **defined**(*identifier*) is intended to supplant the #**ifdef** and #**ifndef** directives. Use of the latter directives is discouraged.

The new directive #**elif** (else-if) is designed for use in #**if** and #**if** **defined** blocks.

14.1 A structure or union can be assigned to another structure or union of the same type. Structures and unions can be passed by value to functions and returned by functions.

In expressions involving $->$, the expression preceding the arrow must have the same type (or be cast to the same type) as the structure to which the member on the right-hand side of the arrow belongs.

17 The listed anachronisms are not recognized.

Appendix B
Syntax Summary

B.1 Tokens 201
B.1.1 Keywords 201
B.1.2 Identifiers 201
B.1.3 Constants 202
B.1.4 Strings 204
B.1.5 Operators 204
B.1.6 Separators 205
B.2 Expressions 205
B.3 Declarations 207
B.4 Statements 210
B.5 Definitions 211
B.6 Preprocessor Directives 211
B.7 Pragmas 212

B.1 Tokens

keyword
identifier
constant
string
operator
separator

B.1.1 Keywords

auto	**double**	**int**	**struct**
break	**else**	**long**	**switch**
case	**enum**	**register**	**typedef**
char	**extern**	**return**	**union**
const*	**float**	**short**	**unsigned**
continue	**for**	**signed**	**void**
default	**goto**	**sizeof**	**volatile***
do	**if**	**static**	**while**

* Not yet implemented

The following identifiers may be keywords in some implementations. See your system documentation for information.

cdecl
far
fortran
huge
near
pascal

B.1.2 Identifiers

identifier:
    letter
    underscore
    identifier letter
    identifier underscore
    identifier digit

letter–one of the following:
a b c d e f g h i j k l m
n o p q r s t u v w x y z
A B C D E F G H I J K L M
N O P Q R S T U V W X Y Z

underscore:
_

digit–one of the following:
0 1 2 3 4 5 6 7 8 9

B.1.3 Constants

constant:
integer-constant
long-constant
floating-point-constant
char-constant
enum-constant

integer-constant:
0
decimal-constant
octal-constant
hexadecimal-constant

decimal-constant:
nonzero-digit
decimal-constant digit

nonzero-digit–one of the following:
1 2 3 4 5 6 7 8 9

octal-constant:
0*octal-digit*
octal-constant octal-digit

octal-digit–one of the following:
0 1 2 3 4 5 6 7

hexadecimal-constant:
0x*hexadecimal-digit*
0X*hexadecimal-digit*
hexadecimal-constant hexadecimal-digit

hexadecimal-digit–one of the following:
0 1 2 3 4 5 6 7 8 9
a b c d e f
A B C D E F

long-constant:
integer-constant l
integer-constant L

floating-point-constant:
fractional-constant exponent
fractional-constant
digit-seq exponent

fractional-constant:
digit-seq . digit-seq
. digit-seq
digit-seq .

digit-seq:
digit
digit-seq digit

exponent:
e *sign digit-seq*
E *sign digit-seq*
e *digit-seq*
E *digit-seq*

sign:
+
–

char-constant:
'char'

char:
rep-char
escape-sequence

rep-char:
Any single representable character except the single quote (’), backslash (\), or new-line character

escape-sequence–one of the following:

\’	\"	\\	*ddd*	\x*ddd*	\a
\b	\f	\n	\r	\t	\v

enum-constant:
identifier

B.1.4 Strings

string-literal:
""
"*char-seq*"

char-seq:
char
char-seq char

B.1.5 Operators

operator–one of the following:

!	~	++	--	+
-	*	/	%	<<
>>	<	<=	>	>=
==	!=	|	&	^
&&	||	=	+=	-=
*=	/=	%=	>>=	<<=
&=	^=	|=	?:	,
[]	( )	.	->	

B.1.6 Separators

separator–one of the following:

```
[   ]   (   )       {   }
*   ,   :   =       ;   #
```

B.2 Expressions

expression:
    identifier
    constant
    string
    expression(*expression-list*)
    expression( )
    expression[*expression*]
    expression.*identifier*
    expression–>*identifier*
    unary-expression
    binary-expression
    ternary-expression
    assignment-expression
    (*expression*)
    (*type-name*)*expression*
    constant-expression

expression-list:
    expression
    expression-list , *expression*

unary-expression:
    unop expression
    sizeof(*expression*)

unop–one of the following:

```
-   ~   !   *   &
```

lvalue:
    identifier
    expression[*expression*]
    expression.*expression*
    expression–>*expression*
    **expression*

(type-name)expression
(lvalue)

type-name:
See Section B.3, "Declarations."

binary-expression:
expression binop expression

binop–one of the following:

*	/	%	+	-
<<	>>	<	>	<=
>=	==	!=	&	|
^	&&	||	,	

ternary-expression:
expression ? *expression* : *expression*

assignment-expression:
lvalue++
lvalue--
++*lvalue*
--*lvalue*
lvalue assignment-op expression

assignment-op–one of the following:

=	*=	/=	%=	+=	-=
<<=	>>=	&=	|=	^=	

constant-expression:
identifier
constant
(type-name)constant-expression
unary-expression
binary-expression
ternary-expression
(constant-expression)

B.3 Declarations

declaration:
    sc-specifier type-specifier declarator-list;
    type-specifier declarator-list;
    sc-specifier declarator-list;
    type-specifier;
    typedef *type-specifier declarator-list*;

sc-specifier:
    auto
    extern
    register
    static

type-specifier:
    char
    double
    enum-specifier
    float
    int
    long
    long int
    short
    short int
    struct-specifier
    typedef-name
    union-specifier
    unsigned
    unsigned char
    unsigned int
    unsigned long
    unsigned long int
    unsigned short
    unsigned short int
    signed
    signed char
    signed int
    signed long
    signed long int
    signed short
    signed short int

enum-specifier:
    enum *tag* { *enum-list*}

enum { *enum-list* }
enum *tag*

tag:
identifier

enum-list:
enumerator
enum-list , *enumerator*

enumerator:
identifier
identifier = *constant-expression*

struct-specifier:
struct *tag* { *member-declaration-list* }
struct { *member-declaration-list* }
struct *tag*

member-declaration-list:
member-declaration
member-declaration-list member-declaration

member-declaration:
type-specifier declarator-list;
type-specifier identifier : *constant-expression*;
type-specifier : *constant-expression*;

declarator-list:
declarator
declarator = *initializer*
declarator-list , *declarator*

declarator:
identifier
range-modifier identifier
declarator[ ]
declarator[*constant-expression*]
**declarator*
declarator( )
declarator(*arg-type-list*)
(*declarator*)

arg-type-list:
type-name
arg-type-list, *type-name*

arg-type-list,...
arg-type-list,
void
void *
...
,

type-name:
type-specifier
type-specifier abstract-declarator

abstract-declarator:
*
*range-modifier**
[ ]
(*arg-type-list*)
* *abstract-declarator*
*abstract-declarator**
abstract-declarator[ ]
abstract-declarator[*constant-expression*]
[ ]*abstract-declarator*
[*constant-expression*]*abstract-declarator*
abstract-declarator( )
abstract-declarator(*arg-type-list*)
(*abstract-declarator*)

initializer:
expression
{ *initializer-list*}

initializer-list:
initializer
initializer-list, *initializer*

typedef-name:
identifier

union-specifier:
union *tag* { *member-declaration-list*}
union { *member-declaration-list*}
union *tag*

range-modifier:
cdecl
far
fortran

huge
near
pascal

B.4 Statements

statement:
break;
case *constant-expression* **:** *statement*
compound-statement
continue;
default : *statement*
do *statement* **while(***expression***);**
*expression***;**
for (⟦*expression*⟧**;**⟦*expression*⟧**;**⟦*expression*⟧**)** *statement***;**
goto *identifier***;**
identifier **:** *statement*
if (*expression***)** *statement* ⟦**else** *statement*⟧
;
return ⟦*expression*⟧**;**
switch (*expression***)** *statement*
while (*expression***)** *statement*

compound-statement:
{⟦*declaration-list*⟧⟦*statement-list*⟧}

declaration-list:
declaration
declaration-list declaration

statement-list:
statement
statement-list statement

B.5 Definitions

definition:
    function-definition
    data-definition

function-definition:
    ⟦*sc-specifier*⟧ ⟦*type-specifier*⟧ *declarator* (⟦*parameter-list*⟧) ⟦*parameter-decs*⟧ *compound-statement*

parameter-list:
    fixed-parameter-list
    variable-parameter-list

fixed-parameter-list:
    identifier
    parameter-list , *identifier*

variable-parameter-list:
    fixed-parameter-list,...
    fixed-parameter-list,
    ...
    ,

parameter-decs:
    declaration
    declaration-list declaration

data-definition:
    declaration

B.6 Preprocessor Directives

directive:
    #
    #**define** *identifier* ⟦(⟦*parameter-list*⟧)⟧⟦*token-seq*⟧
    #**elif** *restricted-constant-expression*
    #**else**
    #**endif**
    #**if** *restricted-constant-expression*
    #**ifdef** *identifier*
    #**ifndef** *identifier*

#include "*string*"
#include <*string*>
#line *digit-seq*
#line *digit-seq string*
#undef *identifier*

token-seq:
token
token-seq token

restricted-constant-expression:
defined (*identifier*)
Any *constant-expression* except for **sizeof**
expressions, casts, and enumeration constants

B.7 Pragmas

pragma:
#pragma *char-seq*

Index

Language Reference Index

! (logical-NOT) operator, 98
!= (inequality) operator, 108
"" (quotation marks)
  See also Double-quote escape sequence; Single-quote escape sequence
  # include directives, used in, 183
  notational conventions, 7
  representation, 14, 192
(number sign), 177
% (remainder) operator, 102
& (address-of) operator, 99
& (bitwise-AND) operator, 110
&& (logical-AND) operator, 112
() (parentheses)
  complex declarators, used in, 48
  expressions, used in, 95
  function calls, used in, 89
  function declarators, used in, 47, 65
  macros, used in, 181
* (indirection) operator, 99
* (multiplication) operator, 102
* (pointer modifier), 47, 64
+ (addition) operator, 104
++ (increment) operator, 117
, (comma)
  argument-type list, used in, 66
  declarations, used in, 54, 65
  function calls, used in, 89, 166
  initialization, used in, 77
  sequential-evaluation operator, 114
– (arithmetic negation) operator, 98
– (subtraction) operator, 104
–– (decrement) operator, 117
–> (arrow) in member-selection expressions, 92
–> (member-selection) operator, 92, 195
. (member-selection) operator, 92
... (three periods), 66
/ (division) operator, 102
< (less-than) operator, 108
<< (left-shift) operator, 107
<= (less-than-or-equal-to) operator, 108
<> (angle brackets), 183
= (simple assignment) operator, 118
== (equality) operator, 108
> (greater-than) operator, 108
>> (right-shift) operator, 107
>= (greater-than-or-equal-to) operator, 108
? : (conditional) operator, 115
[ ] (brackets)
  array declarators, used in, 47, 62
  subscript expressions, used in, 90, 91
[[ ]] (double brackets), 7
^ (bitwise-exclusive-OR) operator, 110
_ (underscore), 23
{ } (braces)
  compound statement, used in, 135, 138
  initialization, used in, 77
| (bitwise-inclusive-OR) operator, 110
|| (logical-OR) operator, 112
~ (bitwise-complement) operator, 98

Abstract declarators, 83
Actual arguments
  conversion, 170
  macro, 179, 181
  order of evaluation, 167
  passing, 169
  pointer, 167, 170
  side effects, 167
  type checking, 170
  variable number, 172
Addition operator (+), 104
Address-of operator (&), 99
Aggregate types
  array, 62
  initialization, 76, 77
  structure, 57
  union, 60
Anachronisms, 195
AND operators
  bitwise (&) 110
  logical (&&) 112
Angle brackets (<>), 183

Apostrophe. *See* Single-quote escape sequence
argc parameter, 32
Argument type checking, 67, 165, 170
  formal parameters, 161
  function calls, 170
Arguments
  See also Parameters
  actual
    conversion, 170
    evaluation, order of, 167
    macro, 179, 181
    passing, 169
    pointer, 167, 170
    side effects, 167
    type checking, 170
    variable number, 172
  command line, 32
  formal. *See* Formal parameters
  main function, 32
  variable number, 66, 172
Argument-type lists, 66, 165
  abstract declarator, used with, 83
  pointer arguments, used with, 67
  void *, used with, 67
  void keyword, used with, 67
argv parameter, 32
Arithmetic conversions, 97, 193
Arithmetic negation operator (–), 98
Array modifier ([ ]), 47, 62
Arrays
  declaration, 47, 62
  elements, 90
  identifiers, 88
  initialization, 76, 77, 80
  multidimensional, 62, 91
  references to, 88, 90
  storage, 62, 92
  subscripts, 90
asm keyword, 191
Assignment
  See also Initialization
  conversions, 124
  described, 87
  expressions, 94
  operators, 116
Associativity
  modifiers, 48
  operators, 120
auto storage class, 69, 72, 75
Backslash character (\ ), 13, 14
Backspace escape sequence, 14
Bell character, 14, 192
Binary
  expressions, 94
  operators, 96
Bit fields, 58, 194
Bitwise-AND operator (&), 110
Bitwise-complement operator (~), 98
Bitwise-exclusive-OR operator (^), 110
Bitwise-inclusive-OR operator (|), 110
Block, 33
Braces ({ })
  compound statement, used in, 135, 138
  initialization, used in, 77
Brackets
  array declarators, used in, 47, 62
  double ([[ ]]), 7
  subscript expressions, used in, 90, 91
Branch statements, 146, 151
break statement, 137
Byte, size of, 193

C character set, 11
Call by reference. *See* Pass by reference
Call by value. *See* Pass by value
Calls. *See* Calls, function
Carriage-return escape sequence, 14
case keyword, 151
Case sensitivity, 12, 23
Casts. *See* Type casts
cdecl keyword, 51, 191
char type
  conversion, 125
  described, 42
  differences from Kernighan & Ritchie, 192
  range of values, 44
  storage, 44
Character constants
  See also Escape sequences
  differences from Kernighan & Ritchie, 192
  form, 20
  type, 21
Character sets, 11
Characters
  backslash (\ ), 13, 14, 15
  backspace escape sequence, 14

Characters *(continued)*
bell, 14, 192
carriage-return escape sequence, 14
case, 12, 23
continuation (\), 15
CONTROL-Z, 12
differences from Kernighan & Ritchie, 192
digits, 12
double-quote escape sequence, 14
end of file, 12
escape sequences, 13
form-feed escape sequence, 14
hexadecimal escape sequence, 14
horizontal-tab escape sequence 14
letters, 12
new-line escape sequence, 14
octal escape sequence, 14
punctuation, 12
single-quote escape sequence, 14
special, 12
vertical-tab escape sequence, 14
white space, 12, 13
Comma (,)
argument-type list, used in, 66
declarations, used in, 54, 65
function calls, used in, 89, 166
initialization, used in, 77
operator, 114
Command-line arguments, 32
Comments, 24
Comparison operators. *See* Relational operators
Compilation, conditional, 184
Complement operators, 98
Complex declarators, 48, 51
Compound assignment operators, 118
Compound statements, 138
Conditional compilation, 184
Conditional operator (?:), 115
Conditional statements, 146, 151
const keyword, 191
Constant expressions
case, 151
conversion, 45
defined(identifier), 186
described, 87
directives, used in, 95, 186
form, 95
initializers, 95
restricted, 95, 186
Constant expressions *(continued)*
switch statement, used in, 151
Constants
character
See also Escape sequences
differences from Kernighan & Ritchie, 192
form, 20
type, 21
conversion, 45
decimal integer, 17, 18
described, 17
enumeration, 56
floating-point, 19, 20, 46
integer
differences from Kernighan & Ritchie, 191
form, 17
hexadecimal
conversion, 19, 46
form, 17
type, 18
long, 19
negative, 18
manifest, 178, 182
octal
conversion, 19, 46
form, 17
type, 18
string. *See* String literals
summarized, 200
type, 88
Continuation character (\), 15
continue statement, 140
CONTROL-Z character, 12
Conventions, notational, 6
Conversions
actual arguments, 170
assignment, 124
constant expressions, 45
constants, 45
enumeration types, 130
floating-point types, 128
formal parameters, 162, 170
function call, 131, 170
hexadecimal constants, 46
implicit, 129
integral types, 129
octal constants, 46
operator, 130
pointer types, 129

Conversions *(continued)*
range of values, effects on, 45
signed integral types, 124
structure types, 130
type cast, 130
union types, 130
unsigned integral types, 126, 129
usual arithmetic, 97, 193
void type, 130

Data types. *See* Types
Decimal integer constants, 17, 18
Declarations
arguments
none, 67
pointer, 67
variable number, 66
form, 41
formal parameters, 160, 161
forward. *See* Declarations, function
function
default return type, 66
default storage class, 75
described, 29, 65, 157, 164
differences from Kernighan & Ritchie, 194
form, 65
implicit, 164
return type, 65, 164
return value, 164
storage class, 74, 164
visibility, 75, 164
pointer, 47, 64, 165
summarized, 205
type, 80, 81
typedef, 80, 82
variable
array, 62
default storage class, 70
described, 29
enumeration, 55
external, 69
form, 53
internal, 69, 72
multidimensional arrays, 62
pointer, 64
simple, 54
structure, 57
union, 60

Declarators
abstract, 83
array, 47
complex, 48
described, 46
function, 47
parentheses, enclosed in, 48
pointer, 47
special keywords, used with, 51
Decrement operator (--), 117
default keyword, 151
Default
return type, 66
storage class
external variable declarations, 70
function declarations, 75
internal variable declarations 73
define directive, 178
defined (identifier) constant expression, 186
Definitions
function
described, 29, 157
storage class, 158
summarized, 209
visibility, 158
variable
described, 29, 70
storage class, 70
summarized, 209
visibility, 70, 73
Digits, 12
Dimensions. *See* Multidimensional arrays
Directives
constant expressions, used in, 95, 186
define, 178
described, 29, 177
differences from Kernighan & Ritchie, 195
elif
described, 185
differences from Kernighan & Ritchie, 195
nesting, 186
else, 185, 186
endif, 185, 186
if, 185, 186, 195
ifdef, 188, 195
ifndef, 188, 195
include, 183

Directives *(continued)*
lifetime, 31
line, 189
restricted constant expressions, 95
summarized, 209
undef, 182
Division operator (/), 102
do statement
described, 141
execution, continuation of, 140
execution, termination of, 137
Double brackets ([[ ]]), 7
Double quote. *See* Quotation marks
double type
conversion, 128
described, 42
internal representation, 46
range of values, 44
storage, 44
Double-quote escape sequence, 14

\ (backslash), 13, 14, 15
Elements, referring to, 90, 91
elif directive
described, 185
differences from Kernighan & Ritchie, 195
nesting, 186
Ellipsis dots, 6
else directive, 185, 186
else keyword, 146
endif directive, 185, 186
End-of-file character, 12
entry keyword, 191
enum type specifier, 55, 191
Enumeration constants, 37, 56
Enumeration expressions, 88
Enumeration set, 55
Enumeration types
conversion, 130
declaration, 55, 81
described, 42
differences from Kernighan & Ritchie, 192
identifiers, 88
range of values, 44
storage, 44, 55
tags
type declarations, 81
variable declarations, 55
tags *(continued)*
naming class, 38
envp parameter, 33
Equality operator (==), 108
Escape sequences
\" escape sequence, 14
\' escape sequence, 14
\\ escape sequence, 14
\a escape sequence, 14
\b escape sequence, 14
\f escape sequence, 14
\n escape sequence, 14
\r escape sequence, 14
\t escape sequence, 14
\v escape sequence, 14
described, 13
differences from Kernighan & Ritchie, 192
Evaluation order, 112, 121
Execution. *See* Program execution
Exit from functions, 149
Exponents, 19
Expression list, 89
Expressions
assignment, 94
binary, 94
case constant, 151
constant. *See* Constant expressions
described, 87
enumeration, 88
evaluation, order of, 121
floating point, 88
function call, 89
grouping, 120
integral, 88
lvalue, 116
member selection, 92, 195
operators, used in, 94
parentheses, enclosed in, 95
pointer, 88
side effects, 123
statements, 142
string literal, 89
structure, 88
subscript, 90, 91
summarized, 203
switch, 151, 194
ternary, 94
type cast, 95
unary, 94
union, 88

extern storage class
  described, 69
  function
    declarations, 74, 164
    definitions, 158
  variables
    external, 69
    internal, 73
External declarations
  described, 69
  function, 74
  variable, 69

far keyword
  conversions, 170
  described, 51
  differences from Kernighan & Ritchie, 191
Fields. *See* Bit fields
__FILE__ identifier, 189
Files
  changing names of, 189
  inclusion of, 183
  nesting of, 184
float type
  conversion, 128
  described, 42
  internal representation, 46
  range of values, 44
  storage, 44
Floating-point
  constants
    form, 19
    internal representation, 46
    negative, 20
  expressions, 88
  identifiers, 88
  types
    conversion, 128
    described, 42
    internal representation, 46
for statement
  continuation of execution, 140
  described, 143
  termination of execution, 137
Formal parameters
  conversion, 162, 170
  declaration, 161
  described, 160
  identifiers, 161
Formal parameters *(continued)*
  macros, 179
  naming class, 37
  storage class, 161
  type checking, 161, 170
Form-feed escape sequence, 14
fortran keyword, 51, 191
Forward declarations. *See* Function declarations
Function
  body, 158, 164
  calls
    argument type checking, 170
    arguments, variable number of, 172
    conversions, 131, 170
    described, 157
    form, 89, 166
    indirect, 167
    pointers, use of, 167
    recursive, 173
declarations
  arguments, 66, 67
  default return type, 66
  default storage class, 75
  described, 29, 65, 157, 164
  differences from Kernighan & Ritchie, 194
  implicit, 164
  return type, 67, 164
  return value, 164
  storage class, 74, 75, 164
  visibility, 75, 164
definitions
  described, 157
  return type, 158
  storage class, 158
  summarized, 209
  visibility, 158
modifier ( ), 47
names. *See* Identifiers
pointers, 165, 167
return type. *See* Return type
Function-call conversions, 131, 170
Function-call expressions, 89
Function prototypes. *See* Argument-type lists
Function type. *See* Return type
Functions
  definitions, described, 29
  exit from, 149

Functions *(continued)*
identifiers, 88
main, 32
naming class, 37
return value, 149

Global
lifetime, 33, 68
variables
described, 34
initialization, 75
references to, 73
visibility, 33
goto statement, 145
Greater-than operator (>), 108
Greater-than-or-equal-to operator (>=), 108
Grouping, 120

Hexadecimal
constants
conversion, 19, 46
differences from Kernighan & Ritchie, 191
form, 17
type, 18
escape sequences, 14, 192
Horizontal-tab escape sequence, 14
huge keyword
conversions, 170
described, 52
differences from Kernighan & Ritchie, 191

Identifiers
array, 88
characters allowed in, 22
differences from Kernighan & Ritchie, 191
enumeration, 88
__FILE__, 189
floating point, 88
formal parameters, 161
function, 88
integral, 88
length, 23
__LINE__, 189
modified, 47

Identifiers *(continued)*
naming classes, 36
pointer, 88
structure, 88
summarized, 199
union, 88
if directive, 185, 186, 195
if statement, 146
ifdef directive, 188, 195
ifndef directive, 188, 195
include directive, 183
Include files, 183, 184
Increment operator (++), 117
Indirection operator (*), 99
Inequality operator (!=), 108
Initialization
arrays, 76, 77, 80
auto storage class, 75
constant expressions, 95
differences from Kernighan & Ritchie, 194
fundamental types, 76
global variables, 75
link time, 71
pointers, 76
register storage class, 75
restrictions, 75
static variables, 75
string literals, 80
structure variables, 76, 77
union variables, 76, 77
Insertion of files, 183
int type
conversion, 126
described, 42
differences from Kernighan & Ritchie, 192
portability, 45
range of values, 44, 45
storage, 44
Integer constants
decimal, 17, 18
differences from Kernighan & Ritchie, 191
hexadecimal, 17, 18, 19
long, 19
negative, 18
octal, 17, 18, 19
Integral
expressions, 88
identifiers, 88

Integral *(continued)*
types
conversion, 124, 126, 129
described, 42
Internal declarations, 69, 72, 73
Internal representation, 45, 46
Italics, 6
Iterative statements
do, 141
for, 143
while, 154

Keywords
differences from Kernighan & Ritchie, 191, 193
listed, 24, 199
notational conventions, 6
special, 51, 64
statements, used in, 135

Labeled statements, 145
Labels
See also Identifiers
case, 151
default, 151
described, 136
form, 145
naming class, 38
Left-shift operator (<<), 107
Less-than operator (<), 108
Less-than-or-equal-to operator (<=), 108
Letters. *See* Characters
Lifetime
described, 33
directives, 30
global, 33, 68
local, 33, 68
Line control, 189
line directive, 189
__LINE__ identifier, 189
Lines, continuation, 15
Lists, linked, 58
Local
lifetime, 33, 68
variables, 34, 164
Logical-AND operator (&&), 112
Logical-NOT operator (!), 98
Logical-OR operator (||), 112
long float type, 42
long type
conversion, 125
described, 42
differences from Kernighan & Ritchie, 192
range of values, 44
storage, 44
Loops
do statement, 141
for statement, 143
while statement, 154
Lvalue expressions, 116

Macros, 178, 179, 181, 182
Main function, 32
Manifest constants, 178, 182
Member-selection expressions, 92, 195
Member-selection operators (-> and .), 92, 195
Members, bit fields, 58
Members
naming class, 38
referring to, 92
structure, 57
union, 60
Modifiers
array, 47, 62
associativity, 48
function, 47
pointer, 47, 64
precedence, 48
Multidimensional arrays, 62, 91
Multiplication operator (*), 102

Names. *See* Identifiers
Naming classes, 36, 194
near keyword
conversions, 170
described, 52
differences from Kernighan & Ritchie, 191
Negation, 98
Nested visibility, 34
New-line escape sequence, 14
Nongraphic escape sequences, 15, 192
NOT operator (!), 98
Notational conventions, 6
Null statement, 148

Number sign (#), 177

Octal
  constants
    conversion, 19, 46
    differences from Kernighan & Ritchie, 191
    form, 17
    type, 18
  escape sequences, 14
One's complement operator (~), 98
Operands, 87
Operators
  addition (+), 104
  address of (&), 99
  arithmetic negation (–), 98
  assignment
    compound, 118
    listed, 116
    simple (=), 118
  associativity, 120
  binary, 96
  bitwise AND (&), 110
  bitwise complement (~), 98
  bitwise exclusive OR (^), 110
  bitwise inclusive OR (|), 110
  complement, 98
  compound assignment, 118
  conditional (?:), 115
  conversions, 130
  decrement (––), 117
  differences from Kernighan & Ritchie, 195
  division (/), 102
  equality (==), 108
  expressions, used in, 94
  increment (++), 117
  indirection (*), 99
  inequality (!=), 108
  left-shift (<<), 107
  listed, 16, 202
  logical AND (&&), 112
  logical
    described, 112
    evaluation, order of, 112
  logical NOT (!), 98
  logical OR (||), 112
  multiplication (*), 102
  one's complement (~), 98
  precedence, 120
Operators *(continued)*
  relational (>,<,<=,>=), 108
  remainder (%), 102
  right shift (>>), 107
  sequential evaluation (,), 114
  shift (<< and >>), 107
  simple assignment (=), 118
  sizeof, 101
  subtraction (–), 104
  ternary (? :), 96, 115
  unary, 96
OR operators
  bitwise exclusive (^), 110
  bitwise inclusive (|), 110
  logical (||), 112
Order of evaluation, 112, 121
Overview, 3

Parameter lists, 160
Parameters
  actual. *See* Actual arguments
  argc, 32
  argv, 32
  envp 33
  formal
    conversion, 162, 170
    declaration, 161
    described, 160
    identifiers, 161
    naming class, 37
    storage class, 161
    type checking, 161, 170
  macro, 179
  main function, 32
Parentheses in
  complex declarators, 48
  expressions, 95
  function calls, 89
  function declarators, 47, 65
  macros, 181
pascal keyword, 51, 191
Pass by
  reference, 170
  value, 166, 169
Pointer modifier (*), 47, 64
Pointers
  adding, 105
  arithmetic, 105
  comparison, 108
  conversion, 129

Pointers *(continued)*
declaration, 47, 64, 165
differences from Kernighan & Ritchie, 193
expressions, 88
function, 165, 167
function calls through, 167
identifiers, 88
implicit conversion, 129
initialization, 76
storage, 64
structure, 64
subtraction, 105
union, 64
Portability, 45
Pound sign (#). *See* Number sign
Pragmas
described, 29, 177
differences from Kernighan & Ritchie, 195
form, 190
summarized, 210
Precedence
modifiers, 48
operators, 120
Predefined identifiers, 189
Preprocessor directives. *See* Directives
Program execution, 32
Program structure, 29
Prototypes. *See* Argument-type lists
Punctuation characters, 12

Quotation marks
See also Double-quote escape sequence; Single-quote escape sequence
include directives, used in, 183
notational conventions, 7
representation, 14, 192

Range of values, 44, 45
Recursion, 173
Reference, passing by, 170
References to global variables, 70, 73
register storage class
described, 73
initialization, 75
internal variables, 72
lifetime, 69
Relational operators (>,<,<=,>=), 108
Remainder operator (%), 102
Removing definitions, 182
Representable character set, 11
Representation, internal, 45, 46
Reserved words. *See* Keywords
Restricted constant expressions, 95, 186
return statement, 149
Return type
declaration, 164
default, 66
described, 67, 158
implicit, 164
Return value, 149, 164
Returning control, 149
Right-shift operator (>>), 107

Search path for include files, 183
Selection statements, 146, 151
Separators, 203
Sequential-evaluation operator (,), 114
Shift operators (<< and >>), 107
short type
conversion, 125
described, 42
differences from Kernighan & Ritchie, 192
range of values, 44
storage, 44
Side effects, 123, 179, 181
signed char type. *See also* char type
signed char type, 42, 194
signed int type. *See also* int type
signed int type, 42
signed keyword, 43, 192
signed long int type. *See also* long type
signed long int type, 194
signed long type. *See also* long type
signed long type, 42, 194
signed short int type. *See also* short type
signed short int type, 42, 194
signed short type. *See also* short type
signed short type, 42, 194
signed type. *See also* int type
signed type, 42, 192
Simple assignment operator (=), 118
Simple variable declarations, 54

Single-quote escape sequence, 14
sizeof operator, 101
Source files, 30
Special characters, 12
Special keywords
  conversions, 170
  declarators, used with, 64
  differences from Kernighan & Ritchie, 191
Standard directories, 183
Statement labels
  described, 136
  form, 145
  naming class, 38
Statements
  body of, 135
  break, 137
  compound, 138
  continue, 140
  do, 141
  expression, 142
  for, 143
  form, 135
  goto, 145
  if, 146
  keywords, used in, 135
  labeled, 136, 145
  listed, 135
  null, 148
  return, 149
  summarized, 208
  switch, 151
  while, 154
static storage class
  described, 69
  function
    declarations, 74, 164
    definitions, 158
  initialization, 75
  variables
    external, 69
    internal, 73
Storage-class specifiers
  auto, 69, 72
  extern
    described, 69
    function declarations, 74, 164
    function definitions, 158
  variables
    external, 69
    internal, 73
Storage-class specifiers *(continued)*
  listed, 69
  register, 69, 73
  static
    described, 69
    function declarations, 74, 164
    function definitions, 158
    variables, external, 69
    variables, internal, 73
Storage classes
  described, 68
  formal parameters, 161
  function declarations, 75, 164
  function definitions, 158
  variable declarations
    external, 70
    internal, 73
Storage
  array types, 62, 92
  bit fields, 58
  char type, 44
  double type, 44
  enumeration types, 44, 55
  float type, 44
  global, 68
  int type, 44, 45
  local, 68
  long type, 44
  pointer types, 64
  short type, 44
  structure types, 58
  union types, 60
  unsigned char type, 44
  unsigned int type, 44, 45
  unsigned long type, 44
  unsigned short type, 44
  void type, 44
String literals
  form, 21, 89
  initializers, 80
  length, 22, 89
  storage, 22
  type, 22
Strings
  See also String literals
  summarized, 202
struct type specifier, 57
Structures
  conversion, 130
  declaration, 57, 81

Structures *(continued)*
differences from Kernighan & Ritchie, 193, 194, 195
expressions, 88
identifiers, 88
initialization, 76, 77
members
bit field, 58
described, 57
naming class, 38
referring to, 92
pointers to, 64
storage, 58
tags
naming class, 38
type declarations, 81
variable declarations, 57
Subscript expressions, 90, 91
Subtraction operator (–), 104
switch statement
constant expressions, used in, 151
described, 151
differences from Kernighan & Ritchie, 194
termination of execution, 137
Symbolic constants. *See* Manifest constants
Syntax conventions. *See* Notational conventions
Syntax summary, 199

Tab escape sequence, 14
Tags
enumeration, 55, 81
naming class, 38
structure, 57, 81
union, 81
Ternary expressions, 94
Ternary operator (?:), 96, 115
Tokens, 25, 199
Transfer statements
break, 137
continue, 140
goto, 145
labeled statements, 145
Two's complement operator, 98
Type
casts, 130
checking. *See* Argument type checking
Type *(continued)*
declarations, 80
names
argument-type lists, used in, 66
described, 83
sizeof, used with, 101
void, 170
specifiers
abbreviations, 43
differences from Kernighan & Ritchie, 192, 193
enum, 42, 55
fundamental types, 42
struct, 57
union, 60
Type-cast expressions, 95
typedef declarations, 80, 82
typedef types, 37, 82
Types
array
declaration, 47, 62
initialization, 76, 77, 80
multidimensional, 62
storage, 62, 92
char
described, 42
storage, 44
conversions. *See* Conversions
differences from Kernighan & Ritchie, 192, 193
double, 42, 44, 46, 128
enumeration
conversion, 130
declaration, 55, 81
described, 42
differences from Kernighan & Ritchie, 192
identifiers, 88
range of values, 44
storage, 44, 55
tags, 38, 55, 81
float
conversion, 128
described, 42
floating-point
conversion, 128
described, 42
internal representation, 46
float
internal representation, 46
range of values, 44

float *(continued)*
storage, 44
function. *See* Return type
fundamental
declaration, 54
described, 42
differences from Kernighan & Ritchie, 192
initialization, 76
listed, 42
range of values, 44
storage, 44
int
conversion, 126
described, 42
differences from Kernighan & Ritchie, 192
integral
conversion, 124, 126, 129
described, 42
int
portability, 45
range of values, 44, 45
storage, 44
long
conversion, 125
described, 42
differences from Kernighan & Ritchie, 192
range of values, 44
storage, 44
pointer
conversion, 129
declaration, 47, 64
implicit conversion, 129
initialization, 76
storage, 64
short, 44
conversion, 125
described, 42
differences from Kernighan & Ritchie, 192
range of values, 44
signed char, 42, 193, 194
signed int, 42
signed long, 42
signed short, 42
structure
conversion, 130
declaration, 57, 81
initialization, 76, 77
structure *(continued)*
pointers to, 64
storage, 58
type names, 83
typedef, 37, 82
union
conversion, 130
declaration, 60, 81
initialization, 76, 77
pointers to, 64
storage, 60
unsigned char
conversion, 126
described, 42
differences from Kernighan & Ritchie, 192, 193, 194
range of values, 44
storage, 44
unsigned int
conversion, 127
described, 42
portability, 45
range of values, 44, 45
storage, 44
unsigned long
conversion, 127
described, 42
differences from Kernighan & Ritchie, 192, 194
range of values, 44
storage, 44
unsigned short
conversion, 127
described, 42
differences from Kernighan & Ritchie, 194
range of values, 44
storage, 44
user defined, 80, 81, 82
void, 42, 44

Unary
expressions, 94
operators, 96
undef directive, 182
Underscore character (_), 23
Union declarations
types, 81
variables, 60
union type specifier, 60

Unions
conversion, 130
declaration, 60, 81
differences from Kernighan & Ritchie, 194, 195
expressions, 88
identifiers, 88
initialization, 76, 77
members
described, 60
naming class, 38
referring to, 92
pointers to, 64
storage, 60
tags, 38, 81
unsigned char type
conversion, 126
described, 42
differences from Kernighan & Ritchie, 192, 193, 194
range of values, 44
storage, 44
unsigned int type
conversion, 127
described, 42
portability, 45
range of values, 44, 45
storage, 44
unsigned keyword, 43, 192
unsigned long int type. *See also* unsigned long type
unsigned long int type, 42, 194
unsigned long type
conversion, 127
described, 42
differences from Kernighan & Ritchie, 192, 194
range of values, 44
storage, 44
unsigned short int type. *See also* unsigned short type
unsigned short int type, 42, 194
unsigned short type
conversion, 127
described, 42
differences from Kernighan & Ritchie, 194
range of values, 44
storage, 44
unsigned type, 42, 192
User-defined types, 80, 81, 82
Usual arithmetic conversions, 97, 193

Value, passing by, 166, 169
Variable names. *See* Identifiers
Variable
declarations
array, 47, 62
described, 29
enumeration, 55
external, 69, 70
form, 53
fundamental types, 54
internal, 69, 72, 73
multidimensional arrays, 62
pointer, 64
simple, 54
structure, 57
summarized, 205
union, 60
visibility, 69
definitions
described, 29, 70
summarized, 209
visibility, 70, 73
Variables
array
declaration, 62
initialization, 77, 80
storage, 62
auto, 69, 72, 75
communal, 70
enumeration, 55
extern, 70, 73
fundamental types, 54, 76
global, 34, 70, 73, 75
local, 34, 164
multidimensional arrays, 62, 91
naming class, 37, 194
pointer, 64, 76
register, 73, 75
simple, 54
static, 70, 73, 75
structure, 57, 58, 77
union, 60, 77
visibility, 69
Vertical-tab escape sequence, 14, 192
Visibility
described, 33
function declarations, 75, 164
function definitions, 158

Visibility *(continued)*
global, 33
nested, 34
variable declarations, 69
variable definitions, 70, 73
void
argument-type list, 66, 67
function-return type, 67
keyword, 191
type name, 170
void *, 67
void type
conversion, 130
described, 42
void type *(continued)*
differences from Kernighan & Ritchie, 192
range of values, 44
storage, 44
volatile keyword, 191

while statement
continuation of execution, 140
described, 154
termination of execution, 137
White-space characters, 12, 13, 192

MICROSOFT®

16011 NE 36th Way, Box 97017, Redmond, WA 98073-9717

Software Problem Report

Name _____

Street _____

City _____ State _____ Zip _____

Phone _____ Date _____

Instructions

Use this form to report software bugs, documentation errors, or suggested enhancements. Mail the form to Microsoft.

Category

_____ Software Problem

_____ Software Enhancement

_____ Documentation Problem
(Document #_____)

_____ Other

Software Description

Microsoft Product _____

Rev. _____ Registration # _____

Operating System _____

Rev. _____ Supplier _____

Other Software Used _____

Rev. _____ Supplier _____

Hardware Description

Manufacturer _____ CPU _____ Memory _____ KB

Disk Size _____" Density: Sides:

Single _____ Single _____

Double _____ Double _____

Peripherals _____

Problem Description

Describe the problem. (Also describe how to reproduce it, and your diagnosis and suggested correction.) Attach a listing if available.

Microsoft Use Only

Tech Support _____ Date Received _____

Routing Code _____ Date Resolved _____

Report Number _____

Action Taken:

Documentation Feedback

Please help us make our documentation better by completing and returning this card. Your feedback will help us improve future products.

Name _____

Address _____

City _____ State _____ Zip _____

Product _____ Version no. _____

Operating system _____

Computer _____ Memory (K) _____

RAM disk ☐ yes ☐ no Hard disk ☐ yes ☐ no

Programming experience: _____ years; in this language: _____years

Please rate the documentation on a scale of 1 to 5.

	5	4	3	2	1	
Complete	☐	☐	☐	☐	☐	Incomplete
Accurate	☐	☐	☐	☐	☐	Inaccurate
Organized	☐	☐	☐	☐	☐	Disorganized
Usable	☐	☐	☐	☐	☐	Unusable
Readable	☐	☐	☐	☐	☐	Unreadable
Understandable	☐	☐	☐	☐	☐	Unintelligible
Attractive	☐	☐	☐	☐	☐	Unattractive
Excellent	☐	☐	☐	☐	☐	Poor

Was installation easy to understand? ☐ yes ☐ no

Did you work through the demonstration session? ☐ yes ☐ no

If yes, was it useful? ☐ yes ☐ no

Is information easy to find? ☐ yes ☐ no

Do you use your documentation mainly for:
☐ reference? ☐ learning to program?
☐ learning to operate the software?

Did you find factual errors in the documentation? ☐ yes ☐ no
If yes, please give manual name, page number, and description of the error.

Is more information needed on how to program? ☐ yes ☐ no
If yes, on what topics?

Is the documentation intimidating? ☐ yes ☐ no

Are the manuals:
☐ too long? ☐ too short? ☐ about the right length?

Other suggestions or comments? (Use back of card if necessary.)

Comments and suggestions become the property of Microsoft Corporation.

detach here

tape here

04/86

MICROSOFT®

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 108 BELLEVUE, WA U.S.A.

POSTAGE WILL BE PAID BY ADDRESSEE

MICROSOFT®
16011 NE 36th Way
Box 97017
Redmond, WA 98073-9717

Attn: Manager, Languages User Education
Bldg. 2

MICROSOFT®